Scientific Computing with Maple Programming

Zhonggang Zeng

August 28, 2000

Contents

Chapter 1

The fundamentals

1.1 Maple as a calculator

Maple can perform many of mathematical calculations with one-line commands. For example:

```
> 3+5*8;
```

$$43$$

```
> 3^5;
```

$$243$$

```
> sqrt(5);
```

$$\sqrt{5}$$

```
> sqrt(5.0);
```

$$2.236067978$$

Notice the difference above between sqrt(5) and sqrt(5.0). When input numbers are integers, Maple performs *simbolic computation* without rounding numbers. If a decimal number is involved, Maple carries out the *numerical computation* with 10 digits precision unless otherwise specified.

The special function "evalf" forces a floating point evaluation:

```
> evalf(  sqrt(5)  );
```

$$2.236067978$$

```
> (1/2)^4; 0.5^4; evalf( (1/2)^4 );
```

$$\frac{1}{16}$$
$$.0625$$
$$.06250000000$$

Readers can see the differences above.

Maple does not execute the command until getting the semicolon ";" or colon ":". If a colon is used, the result of the command will not appear in the worksheet.

Maple accepts definitions of variables with :=

```
> x:=2; y:=5;
```

$$x := 2$$
$$y := 5$$

```
> (x+3*y)/(2*x-7*y);
```

$$\frac{-17}{31}$$

```
> evalf(%);
```

$$-.5483870968$$

The sign % represents the most resent result. In the example above, it represents $\frac{-17}{31}$. As mentioned before, the function evalf calculates the decimal number result.

Maple accepts Greek letters

```
> alpha,  beta,  gamma,  Alpha,  Beta,  Gamma;
```

$$\alpha, \quad \beta, \quad \gamma, \quad A, \quad B, \quad \Gamma$$

However, *the simbol "Pi" carries the value of the special number*

$$\pi = 3.14159265358979323846 26\cdots$$

while "pi" does not.

```
> evalf(Pi);
```

$$3.141592654$$

```
> alpha:=2*Pi/3;
```

$$\alpha := \frac{2}{3}\pi$$

```
> sin(alpha), cos(alpha), tan(alpha), cot(alpha);
```

$$\frac{1}{2}\sqrt{3}, \quad \frac{-1}{2}, \quad -\sqrt{3}, \quad -\frac{1}{3}\sqrt{3}$$

```
> evalf(%);
```

$$.8660254040, -.5000000000, -1.732050808, -.5773502693$$

```
> ln(alpha), exp(alpha);
```

$$\ln(\frac{2}{3}\pi), \quad e^{(2/3\,\pi)}$$

```
> evalf(ln(alpha)), evalf(exp(alpha));
```

$$.7392647780, 8.120527402$$

Functions are defined as "operators". Once defined, they can be calculated or manipulated.

```
> f:=x->sin(x)/x;
```

$$f := x \to \frac{\sin(x)}{x}$$

```
> f(alpha), f(1.57);
```

$$\frac{\sin(\alpha)}{\alpha}, .6369424732$$

```
> f(a+b);
```

$$\frac{\sin(a+b)}{a+b}$$

Taking derivative:

```
> D(f);
```

$$x \to \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2}$$

Table 1.1 is a list of frequently used mathematical operations and their corresponding maple symbols and syntax.

If an arithmetic expression consists of several operations and function evaluations, Maple follows common precedence of operations:

1. Exponentiation or function evaluation (highest)

2. Multiplication or division

3. arithemetic negation

4. Addition and/or subtraction (lowest)

For complicated mathematical expressions, we can use parentheses ( ) to override normal precedences.

Operation	symbol	Example syntax	Example result
Addition	$+$	$3+5$	8
Subtraction	$-$	$3-5$	-2
Multiplication	*	3*5	15
Division	/	$3/5$	$\frac{3}{5}$
power	^	3^5	3^5
exponential function	exp	exp(3.2)	$e^{3.2}$
square root	$\sqrt{\ }$	sqrt(5)	$\sqrt{5}$
sine	sin	sin(3*Pi/4)	$\sin(\frac{3\,\pi}{4})$
cosine	cos	cos(Pi/6)	$\cos(\frac{\pi}{6})$
tangent	tan	tan(2*Pi/3)	$\tan(\frac{2\,\pi}{3})$
cotangent	cot	cot(x)	$\cot(x)$
natural logorithm	ln	ln(2.3)	.8129091229
factorial	!	5!	120

Table 1.1: Common operations

Warning: Although brackets [ ] and braces { } are legitimate substitutes for parentheses in mathematical writing, they are not allowed in Maple to override precedences. Brackets are used for data type *list* and braces are used for data type *set*.

Examples:

Notice the difference between **a+b/c+d** and **(a+b)/(c+d):**

```
> a+b/c+d;
```

$$a+\frac{b}{c}+d$$

```
> (a+b)/(c+d);
```

$$\frac{a+b}{c+d}$$

Let's see some complicated expressions. Pay attention to the use of parentheses.

```
> (a/(b+c) - d^2)/(3*e);
```

$$\frac{1}{3}\,\frac{\frac{a}{b+c}-d^2}{e}$$

The expression $\left(1+\frac{3\,P}{h^2}\right)\left[1-\left(\frac{a}{l}\right)^{.6}\right]$ should be entered as follows

```
> (1+(3*P)/(h^2))*(1-(a/l)^0.6);
```

$$\left(1+3\,\frac{P}{h^2}\right)\left(1-\left(\frac{a}{l}\right)^{.6}\right)$$

More examples:

```
> c*a1*a2*sqrt(g*(h1-h2)/s)/(3+sqrt(a1^2-a2^2));
```

$$\frac{c\,a1\;a2\,\sqrt{\frac{g\,(h1-h2)}{s}}}{3+\sqrt{a1^2-a2^2}}$$

```
> 2*Pi*epsilon/arccos(a^2+b^2-d^2/(2*a*b));
```

$$2\,\frac{\pi\,\varepsilon}{\arccos\left(a^2+b^2-\frac{1}{2}\,\frac{d^2}{a\,b}\right)}$$

1.2 Simple programming

A computational task, in general, involve three components:

1. Data,
2. computational method,
3. results.

As a simple example, we have a formula for the volume of a cylinder:

$$\text{volume of a cylinder} = \text{base area} \times \text{height}.$$

If the base of the cylinder is a circular disk with radius r, then

$$\text{base area} = \pi\, r^2.$$

To compute the volume of the cylinder from the radius r and height h, we have

Data:	r and h
method:	step 1: base area $s = \pi\, r^2$
	step 2: volume $v = s\, h$
	(or combination of step 1 and 2: $v = \pi\, r^2\, h$)
result:	volume v

A program is an implementation of the computational method for the task. It accepts data as *input*, performs the method, and transmits results out as *output*, as shown in the diagram:

A program works as a "black box" for end uses, who enter data as input and activate the program, then see the results.

Using the example of volumn computation for cylinders,, we can write a simple program into Maple worksheet as follows. To break a line before the end of the program, use Shift-Enter instead of Enter key. If no apparent errors, Maple echoes with the program after hitting the semi-colon.

```
> cylinder_volume:=proc(radius,height)  # program definition
     local base_area, volume;           # local variables

     base_area:=Pi*radius^2;            # implement method
     volume:=evalf(base_area*height);

     print(`The volume`, volume);       # output result

  end;
```

$$
\begin{array}{l}
\textit{cylinder\_volume} := \textbf{proc}(\textit{radius}, \textit{height}) \\
\textbf{local}\, \textit{base\_area}, \textit{volume}; \\
\quad \textit{base\_area} := \pi \times \textit{radius}^2\,; \\
\quad \textit{volume} := \mathrm{evalf}(\textit{base\_area} \times \textit{height})\,; \\
\quad \mathrm{print}(`\textit{The volume}`, \textit{volume}) \\
\textbf{end}
\end{array}
$$

The program is named *cylinder_volume*, which accepts arguments *radius* and *height* as input. Local variables (i.e. variables only used inside the program) *base_area* and *volume* are defined. Then the computation is carried out by computing the values of base_area and volume. Finally, the result will be shown using the command "print".

Suppose the radius is 5.5m and the height is 8.2m. We execute the program under Maple:

```
> r:=5.5; h:=8.2;
```

$$r := 5.5$$
$$h := 8.2$$

```
> cylinder_volume(r,h);
```

$$\textit{The volume}, 779.2720578$$

A program is designed to be used again and again. Every time we use it, all we need is to supply the input data according to the rule of the program. In this example, the rule is that radius first and height second.

For radius 3 ft and height 2 ft:

```
> cylinder_volume(3,2);
```

$$\textit{The volume}, 56.54866777$$

When order is reversed, the answer is different.

```
> cylinder_volume(2,3);
```

$$\textit{The volume}, 37.69911185$$

In summary, a maple program generally consists of the following components:

1. Program definition statement in the form of

   $$\textit{program\_name}{:=}\mathrm{proc}(\textit{argument\_list})$$

   Users will use *program_name* to activate the program. The list of variables in *argument_list* is the entrance for input data. *Do not end this line with semicolon ";" if there is local variable declaration line to be described below.*

2. Local variable declaration in the form of

   local *variable_list*;

   This declaration is part of program definition statement.

3. Maple statements that carry out the computational method, using the arguments representing input data.

4. Output mechanism. The simplest way is print statement.

5. closing statement, i.e. the line "end;"

6. Comments. Starting with the character #, the remainder of the line is considered comments by the programmer with no effect on the program execution.

Although you can directly type programs into Maple worksheet, It is prefered to separate the process of programming into 4 steps: (1) ***editing***, (2) ***loading***, (3) ***test run and revision***, (4) ***execution***.

(1) ***Editing***:

Use your favourate text editor, such as Notepad, to edit your program as a *text file* with filename, say "file.txt". Make sure to save it on the floppy disk (usually "a:" drive).

Remark: A DOS filename consists of {*filename*}.{*extention*}. There must be no more than 8 characters in the *filename* part and no more than 3 characters in the *extention* part. The extention of the filename is used to indicate the type of the file. The following are some common extentions:

.txt —- text file

.mws —- maple worksheet

.htm —- HTML file (for World Wide Web use)

.exe —- DOS executable file

Although Windows 95/98 accepts longer filenames in more flexible style, it is recommended that the original DOS restriction be followed.

(2) ***Loading:***

At Maple prompt ">", type

>read(‘a:file.txt‘);

Maple then starts reading the text file. If Maple finds a mistake while reading, it will stop and give you an error message. Otherwise, your program will be loaded into the Maple worksheet.

Remark: The two single quotes ‘ used in the command above is usually on the upper-left corner of the keyboard below ESC key. A common mistake is using the quote ’ next to the Enter key.

(3) ***Test run and revision***

If Maple give you an error message, or you suspect a wrong result after a test run, you should go back to Notepad to revise your program

(4) ***Execution***

See example below

Example 1: The future value of an annuity: Suppose you make a *periotic deposit* \$R to an account that pays *interest rate i per period*, for *n periods*, then the *future value* \$S of the account after that n periods will be

$$S = \frac{R\left((1+i)^n - 1\right)}{i}.$$

Write a program to calculate the worth of a retirement plan at the end of the term if the plan pays an *annual rate A*, requires *monthly* deposit \$R, for y *years.* Then use the program to calculate the future value if the monthly deposit is \$300, the annual interest rate is 12%, for 30 year.

Solution: First, we see that the computational task involve input items:

R — deposit

A — annual rate (instead of i)

n — the number of years

We type the following program using Notepad and save as *a:future.txt.* Figure 1.1 shows the program text file.

Remarks:

```
# Program that compute the future value of an annuity
# Input: R: monthly deposit (\$)
#        A: annual interest rate (0.12 for 12\%)
#        y:  number of years
#
future:=proc(R,A,y)             # defines the program "future"
                                #    and arguments R, I, y
   local  i, n, S;              #  define local variables

   i:=A/12;                     # monthly interest rate is
                                #    1/12 of the annual rate
   n:=y*12;                     # totoal number of months
   S:=R*((1+i)\symbol{94}n-1)/i;    # calculation of future value

   print(`The future value`, S); # print the result
end;                            # the last line of the program.
```

Figure 1.1: Future value program

(1) Don't put the semi-colon at the end of proc(...). it is finished at the end of local ....

(2) At end of every definition or action inside the program, remember that you need ";".

(3) Especially, type ";" after *end*.

(4) You can, and should, put comments in your program using "#", as above.

Now read into maple.

```
> read(`a:future.txt`);
```

$$
\begin{array}{l}
\mathit{future} := \mathbf{proc}(R,\,A,\,y) \\
\mathbf{local}\, i,\, n,\, S; \\
\quad i := 1/12 \times A\,;\, n := 12 \times y\,;\, S := R \times \left((1+i)^n - 1\right)/i\,; \\
\quad \mathrm{print}(`\mathit{The\ future\ value}`,\, S) \\
\mathbf{end}
\end{array}
$$

Define the input:

```
> Deposit:=300; AnnualRate:=0.12; Years:=30;
```

$$\mathit{Deposit} := 300$$

$$\mathit{AnnualRate} := .12$$

$$\mathit{Years} := 30$$

Now execute the program with the defined input above. Notice that the variables R, A, y in the program are "dummy" variables. When you execute the program, you just substitute them with variables carrying the data.

```
> future(Deposit,AnnualRate,Years);
```

$$\mathit{The\ future\ value},\ .1048489240\,10^7$$

Or, you may directly input numbers

```
> future(300, .12, 30);
```

$$\mathit{The\ future\ value},\ .1048489240\,10^7$$

Example 2: A program that compute the two solutions of the general quadratic equation

$$ax^2 + bx + c = 0$$

with the quadratic formula. Use the program to solve

$$3\,x^2 - 5\,x + 2 = 0 \quad \text{and} \quad x^2 - 3\,x + 2 = 0$$

Solution: Type the following program, shown below, using Notepad and save as file *a:quadsolv.txt*

```
quad:=proc(a,b,c)                          # define the program "quad"
                                           #   with arguments a, b, c
   local sol1, sol2;                       # define local variables
                                           # calculate sol1, sol2
   sol1:=(-b+sqrt(b\symbol{94}2-4*a*c))/(2*a);
   sol2:=(-b-sqrt(b\symbol{94}2-4*a*c))/(2*a);

   print( `The solutions`,sol1, sol2);  # print results

end;                                       # end of program
```

Now load program

```
> read(`a:quadsolv.txt`);
```

$$\begin{array}{l} \mathit{quad} := \mathbf{proc}(a,\,b,\,c) \\ \mathbf{local}\ \mathit{sol1},\ \mathit{sol2}; \\ \quad \mathit{sol1} := 1/2 \times (-b + \mathrm{sqrt}(b^2 - 4 \times a \times c))\big/a\,; \\ \quad \mathit{sol2} := 1/2 \times (-b - \mathrm{sqrt}(b^2 - 4 \times a \times c))\big/a\,; \\ \quad \mathrm{print}(`\mathit{The\ solutions}`,\ \mathit{sol1},\ \mathit{sol2}) \\ \mathbf{end} \end{array}$$

To solve $3\,x^2 - 5\,x + 2 = 0$, notice that $a = 3$, $b = -5$, and $c = 2$:

```
> quad(3,-5,2);
```

$$\mathit{The\ solutions},\ 1,\ \frac{2}{3}$$

To solve $x^2 - 3\,x + 2 = 0$

```
> a:=1; b:=-3; c:=2;
```

$$a := 1$$
$$b := -3$$
$$c := 2$$

```
> quad(a,b,c);
```

$$\mathit{The\ solutions},\ 2,\ 1$$

Example 3: The principal stresses (whatever they mean) are given by

$$S_{max} = A + B, \quad S_{min} = A - B$$

where

$$A = E\,1 - \frac{\mathrm{K}(Q_1 + Q_3)}{2\,(1-\mu)}$$

$$B = \frac{E\,(1+K)\,\sqrt{(Q_1 - Q_2)^2 + (Q_2 - Q_3)^2}}{2\,(1+\mu)}$$

Write a program that will accept input for E, μ, K, Q_1, Q_2, Q_3 and print the values of the principal stresses.

Sample Data

E	μ	K	Q_1	Q_2	Q_3
9.3×10^6	0.32	0.05	138	-56	786

Solution: We write the program

```
principal_stress:=proc(E, mu, K, Q1, Q2, Q3)
   local Smax, Smin, A, B;

   A:=E*(1-K)*(Q1+Q3)/(2*(1-mu));

   B:=E*(1+K)*sqrt((Q1-Q2)\symbol{94}2+(Q2-Q3)\symbol{94}2)/(2*(1-mu));
```

```
    print(`S\_max,  S\_min`);
    print(A+B,A-B);

end;
```

```
> read(`a:prnstres.txt`);
```

$principal\_stress := \mathbf{proc}(E, \mu, K, Q1, Q2, Q3)$
$\mathbf{local}\, Smax, Smin, A, B;$
$A := E \times (1 - K) \times (Q1 + Q3) \big/ (2 - 2 \times \mu)\,;$
$B := E \times (1 + K) \times \mathrm{sqrt}((Q1 - Q2)^2 + (Q2 - Q3)^2) \big/ (2 - 2 \times \mu)\,;$
$\mathrm{print}(`S\_max,\ \ S\_min`)\,;$
$\mathrm{print}(A + B, A - B)$
$\mathbf{end}$

```
> principal_stress(9.3*10^6, 0.32, 0.05, 138.0, -56.0, 786.0);
```

$$S\_max, \quad S\_min$$

$$.1220668212\, 10^{11}, \; -.201476241\, 10^{9}$$

1.3 Conditional statements

Example 1. A piecewise function

$$\mathrm{f}(x) = \begin{cases} -(x-1)^2 + 2 & x < 0 \\ 2\,x + 1 & 0 \le x \le 2 \\ 5\,e^{(-(x-2)^2)} & x > 2 \end{cases}$$

can be calculated with a maple program (edited using Notepad with filename a:func.txt)

```
func:=proc(x)            # definition
                         # no local variables needed
   if x< 0 then
      -(x-1)^2+2;        # case for x < 0
   elif x<2 then
      2*x+1;             # case for 0 <= x <= 2
   else
      5*exp(-(x-2)^2);   # case for x > 2
   fi;

end;
```

```
> read(`a:func.txt`);
```

$func := \mathbf{proc}(x)$
$\quad \mathbf{if}\, x < 0\, \mathbf{then}\, -(x-1)^2 + 2\, \mathbf{elif}\, x \le 2\, \mathbf{then}\, 2 \times x + 1\, \mathbf{else}\, 5 \times \exp(-(x-2)^2)\, \mathbf{fi}$
$\mathbf{end}$

```
> func(-3); func(1); func(3);
```

$$-14$$
$$3$$
$$5\, e^{(-1)}$$

Program note: When executing a Maple program on Maple worksheet, the result of the very last executed statement is automatically shown without the need to use "print". For example, if input x is negative, then $-(x-1)^2+2$ is the last statement executed and its result is "printed" automatically.

An if-block has the following structures:

1. Statements executed only if the condition is met. Otherwise, the statements will be skipped:

   if *condition* then
   statements
   fi;

2. There are two cases separated by one condition: the condition is met or else:

   if *condition* then
   statement block 1
   else
   statement block 2
   fi;

   Entering this if-block, the computer verifies the *condition*, if the answer is "true", the *statement block 1* will be executed and the *statement block 2* will be skipped. On the other hand, if the answer to the *condition* is "false", the computer will skip the *statement block 1* and execute *statement block 2*

3. There are many cases with many conditions:

   if *condition 1* then
   statement block 1
   elif *condition 2* then
   statement block 2
   ...
   ...
   elif *condition n* then
   statement block n

else
   final statement block
fi;

Entering this if-block, the computer verifies the *condition 1* first. If the answer is "false", the machine verifies the *condition 2*, and continue until *condition k* returns "true". In that case the *statement block k* will be executed and all other statements will be skipped. If the answers to all the *conditions* are "false", the computer execute *the final statement block* and skip all others. ("else" and *final statement block* are optional).

Remarks:

- Do not put semicolon at end of "then" or "else".
- It is a good practice to align those "if", "elif", "else" and "fi" of the same if block, and have a 3-space indentation for each statement block.
- All the conditions must be verifiable by Maple. For example, Maple cannot verify if $\sqrt{5} > 1$ and will return a "boolean error", which is common in Maple programming

Example 2. Suppose in a certain course, after five 100-point exams, the course grade will be determined by the scale: A for 90% or above, B for 80-89.9%, C for 70-79.9%, D for 60-69.9% and F for 0-59.9%. Write a program that, for input of total points, print (1) the course grade, (2) a borderline warning if the points is within 1% of next higher grade.

Solution: the program (filename: a:grade.txt)

```
grade:=proc(points)

   if points >= 450 then
      print(' A ');
   elif points >=400 then
      print(' B ');
      if points 445 then
         print('borderline A');
      fi;
   elif points >= 350 then
      print(' C ');\\
      if points > 395 then\\
         print('borderline B');\\
      fi;
   elif points >= 300 then\\
      print(' D ');\\
      if points > 345 then\\
         print('borderline C');\\
      fi;
  else
```

```
            print(' F ');\\
            if points > 295 then\\
               print('borderline D');\\
            fi;
        fi;

     end;
```

```
> read('a:grade.txt');
```

grade := **proc**(*points*)
  if $450 \leq points$ **then** print(' A ')
  elif $400 \leq points$ **then** print(' B ') ; **if** $445 < points$ **then** print('*borderline* A') **fi**
  elif $350 \leq points$ **then** print(' C ') ; **if** $395 < points$ **then** print('*borderline* B') **fi**
  elif $300 \leq points$ **then** print(' D ') ; **if** $345 < points$ **then** print('*borderline* C') **fi**
  else print(' F ') ; **if** $295 < points$ **then** print('*borderline* D') **fi**
  fi
end

```
> grade(367);
```

$$C$$

```
> grade(397);
```

$$C$$

borderline B

```
> grade(310);
```

$$D$$

Example 3. (Techniques in this example will be used in probability simulations). Suppose we number a standard deck from 1 to 52, such that

hearts: 1-13; spades: 14-26, diamonds: 27-39, clubs: 40-52

in each suit, cards are numbered in the following order:

ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king

Write a program that for input of card name (e.g. 3, spade, or jack, diamond), print the number associated with the card.

Remark on data type: Maple classifies data in many types. (try ?type). We need to specify types *numeric* and *string* in this problem. Card value 2, 3, ..., 10 are numeric type. A sequence of characters within a pair of double quote " " is called a string. A string carries no value other than itself. For example, "ace", "jack", "queen", and "king" are string type. In comparison, ace, jack, queen and king without quotes are variable names. The function type(expression, type name) will return either "true" or "false" depending on whether the expression matches the type name. For example:

```
> type(3.5,string);
```

false

```
> type(3.5,numeric);
```

true

```
> type(queen, string);
```

false

```
> type("queen",string);
```

true

```
> Value:=6:      type(Value,numeric);
```

true

```
> a:="king":  type(a,string);
```

true

Now the program:

```
# program that numbers a card in a deck from 1-52
#   input:
#      value  --- one of the following:
#                   ace, 2, 3, ..., 10, jack, queen, king
#      suit   --- one of the following
#                    "heart", "spade", "diamond", "club"
#
#   output:   the number of the card (1-52)
#
cardnum:=proc(value,suit)

  local order, number, error;

  error:=false;
  if type(value,numeric) then
     order:=value;
  elif type(value,string) then
     if value="ace" then
        order:=1;
     elif value="jack" then
        order:=11;
     elif value="queen" then
        order:=12;
     elif value="king" then
        order:=13;
     else
        print(`Error: wrong value for the first argument`);
        error:=true
     fi
  else
     print(`Error: wrong data type for first argument`);
     error:=true;
  fi;

  if suit="heart" then
     number:=order;
  elif suit="spade" then
```

```
       number:=13+order;
    elif suit="diamond" then
       number:=26+order;
    elif suit="club" then
       number:=39+order;
    else
       print(`Error: wrong value for the second argument`);
       error:=true;
    fi;

    if error=false then
       print(`The card number`, number);
    fi;

  end;
```

```
> read(`a:cardnum.txt`);
```

$cardnum := \mathbf{proc}(value,\ suit)$
$\mathbf{local}\ order,\ number,\ error;$
$\quad error := false\ ;$
$\quad \mathbf{if}\ \mathrm{type}(value,\ numeric)\ \mathbf{then}\ order := value$
$\quad \mathbf{elif}\ \mathrm{type}(value,\ string)\ \mathbf{then}$
$\quad\quad \mathbf{if}\ value = \text{“ace”}\ \mathbf{then}\ order := 1$
$\quad\quad \mathbf{elif}\ value = \text{“jack”}\ \mathbf{then}\ order := 11$
$\quad\quad \mathbf{elif}\ value = \text{“queen”}\ \mathbf{then}\ order := 12$
$\quad\quad \mathbf{elif}\ value = \text{“king”}\ \mathbf{then}\ order := 13$
$\quad\quad \mathbf{else}\ \mathrm{print}(\text{`}Error:\ wrong\ value\ for\ the\ first\ argument\text{`});\ error := true$
$\quad\quad \mathbf{fi}$
$\quad \mathbf{else}\ \mathrm{print}(\text{`}Error:\ wrong\ data\ type\ for\ first\ argument\text{`});\ error := true$
$\quad \mathbf{fi};$
$\quad \mathbf{if}\ suit = \text{“heart”}\ \mathbf{then}\ number := order$
$\quad \mathbf{elif}\ suit = \text{“spade”}\ \mathbf{then}\ number := 13 + order$
$\quad \mathbf{elif}\ suit = \text{“diamond”}\ \mathbf{then}\ number := 26 + order$
$\quad \mathbf{elif}\ suit = \text{“club”}\ \mathbf{then}\ number := 39 + order$
$\quad \mathbf{else}\ \mathrm{print}(\text{`}Error:\ wrong\ value\ for\ the\ second\ argument\text{`});\ error := true$
$\quad \mathbf{fi};$
$\quad \mathbf{if}\ error = false\ \mathbf{then}\ \mathrm{print}(\text{`}The\ card\ number\text{`},\ number)\ \mathbf{fi}$
$\mathbf{end}$

```
> cardnum(3,"diamond");
```

$$The\ card\ number,\ 29$$

```
> cardnum("king","club");
```

$$The\ card\ number,\ 52$$

```
> cardnum(queen,club);
```

$$Error:\ wrong\ data\ type\ for\ first\ argument$$
$$Error:\ wrong\ value\ for\ the\ second\ argument$$

```
> cardnum("queen","spade");
```

$$The\ card\ number,\ 25$$

1.4 Loops with "do" statements

1.4.1 Structure with for–do

Repetitive tasks are carried out by *loops*. For example, if we want to square integer i, $i = 1, 2, 3, \cdots, 10$, the following statements, called a loop, does the job.

```
> for i from 1 to 10 do

> print(i^2)

> od;
```

$$1$$
$$4$$
$$9$$
$$16$$
$$25$$
$$36$$
$$49$$
$$64$$
$$81$$
$$100$$

The simplest loop structure is

> **for** *loop_index* **from** *start_value* **to** *end_value* **do**
> *block of statements to be repeated*
> **od;**

We present, in the list below, the general rules and concepts of for–do loops.

1. The *loop_index* is a variable name such as i, j.
2. *start_value* and *end_value* can be numbers, variable name carrying numbers, or executable expressions resulting in numbers, such as 1, 10, $k1$, $k2$, $2 * n$, etc. The numbers they represent are usually integers, while decimal numbers or even fractions are allowed.
3. The loop works in the following way:
   (a) Assign the *start_value* to *loop_index* .
   (b) Comparing the *loop_index* value with *end_value* , if *loop_index* > *end_value* , jump out of the loop to the line following "od;". Otherwise go to the *block of statements to be repeated.*

(c) After confirming that *loop_index* $<$ *end_value*, execute *block of statements to be repeated*.

(d) Add 1 to *loop_index* and go back to (b).

4. Do not alter the value of *loop_index*. Let the loop handle it instead.

5. If the *start_value*=1, then "**from** *start_value*" part may be omitted. That is, the loop in the above example can be equivalently written as

```
for i to 10 do
  i^2;
od;
```

6. Do not put semicolon ";" at the end of "do". Semicolon or colon should be at the end of "od".

7. The *loop_index* increase by the default stepsize 1 Every time the execution of *block of statements to be repeated* is finished. Different stepsize can be used by adding "by *stepsize*" feature. For example,

```
for i from 20 to 2 by -2 do
  i^2;
od;
```

produces 20^2, 18^2, 16^2, $\cdots$, 4^2, 2^2.

As an example, the following do loop calculates sine and cosine function from 0 to π by $\frac{\pi}{5}$.

```
> for t from 0 by evalf(Pi/5) to evalf(Pi) do
     s1:=evalf(sin(t));
     s2:=evalf(cos(t));
  od;
```

$$s1 := 0$$
$$s2 := 1.$$
$$s1 := .5877852524$$
$$s2 := .8090169943$$
$$s1 := .9510565165$$
$$s2 := .3090169938$$
$$s1 := .9510565160$$
$$s2 := -.3090169952$$
$$s1 := .5877852514$$
$$s2 := -.8090169950$$

1.4.2 An introduction of simple arrays

An array is used to store a sequence of data. For example, when we divide an interval [0,1] into 100 subintervals of equal length 0.01 with points (called nods)

$$x_0 = 0,\, x_1 = .01,\, x_2 = .02,\, \cdots,\, x_{99} = .99,\, x_{100} = 1.00$$

we can use an array to carry these nods.

First, open an (empty) array

```
> nod:=array(0..100);
```

$$nod := \mathrm{array}(0..100, [])$$

Then we generate the nods with a loop.

```
> for i from 0 to 100 do
     nod[i]:=i*0.01
  od:
```

The values of x_i for each i is stored in nod[i] for i from 0 to 100. We can show the values using the sequence command seq:

```
> seq(nod[i],i=0..100);
```

0, .01, .02, .03, .04, .05, .06, .07, .08, .09, .10, .11, .12, .13, .14, .15, .16, .17, .18, .19, .20, .21, .22, .23, .24, .25, .26, .27, .28, .29, .30, .31, .32, .33, .34, .35, .36, .37, .38, .39, .40, .41, .42, .43, .44, .45, .46, .47, .48, .49, .50, .51, .52, .53, .54, .55, .56, .57, .58, .59, .60, .61, .62, .63, .64, .65, .66, .67, .68, .69, .70, .71, .72, .73, .74, .75, .76, .77, .78, .79, .80, .81, .82, .83, .84, .85, .86, .87, .88, .89, .90, .91, .92, .93, .94, .95, .96, .97, .98, .99, 1.00

Each member of the array is referenced by array_name[index]. In this example,

```
> nod[19];
```

$$.19$$

```
> nod[87];
```

$$.87$$

1.4.3 Iteration

Example 1: Write a program to generate the first n terms of the the sequence

$$x_k = \frac{x_{k-1}}{2} + \frac{1}{x_{k-1}}, \quad x_0 = 1, \quad k = 1, 2, \cdots, n$$

The sequence approaches $\sqrt{2}$.

Solution: The program requires an input item n. We edit the program as file `a:sqrt2.txt`

```
#
# Program that generates a sequence
# converging to square root of 2
#
# input   n --- ending index of the sequence
#
sqrt2:=proc(n)
   local x, k;

   x:=array(0..n); # define the array
   x[0]:=1;        # initialize the 0-th entry of the array

                   # loop genreating the remaining entries
   for k from 1 to n do
      x[k]:=evalf( x[k-1]/2 + 1/x[k-1] );
   od;

   print( seq( x[k], k=0..n ) )  # output

end;
```

```
> read(`a:sqrt2.txt`);
```

$$\begin{array}{l} \mathrm{sqrt2} := \mathbf{proc}(n) \\ \mathbf{local}\, x,\, k; \\ \quad x := \mathrm{array}(0..n)\,; \\ \quad x_0 := 1\,; \\ \quad \mathbf{for}\, k\, \mathbf{to}\, n\, \mathbf{do}\, x_k := \mathrm{evalf}(1/2 \times x_{k-1} + 1\big/x_{k-1})\, \mathbf{od}\,; \\ \quad \mathrm{print}(\mathrm{seq}(x_k,\, k = 0..n)) \\ \mathbf{end} \end{array}$$

```
> sqrt2(10);
```

1, 1.500000000, 1.416666667, 1.414215686, 1.414213562, 1.414213562, 1.414213562, 1.414213562, 1.414213562, 1.414213562, 1.414213562

We can see that the sequence converges to $\sqrt{2}$ very fast.

Example 3. Write a program that generate first n terms of the Fibonacci sequence

$$F_0 = 0, \;\; F_1 = 1, \;\; F_k = F_{k-1} + F_{k-2}, \quad \text{for} \quad k = 2, 3, \cdots, n$$

Solution:

```
# program that generate first n terms of the Fibonacci sequence.
```

```
# input: n (n>=2 ).
#
Fibonacc:=proc(n)
   local k, F;

   F:=array(0..n);          # define the array
   F[0]:=0;  F[1]:=1;       # the first two terms

   for k from 2 to n do     # iterate
      F[k]:=F[k-1]+F[k-2]
   od;

   seq( F[k], k=0..n )      # output
end;
```

The test run

```
> read(`a:fibon.txt`);
```

$$
\begin{array}{l}
\mathit{Fibonacc} := \mathbf{proc}(n) \\
\mathbf{local}\, k,\, F; \\
\quad F := \mathrm{array}(0..n)\,; \\
\quad F_0 := 0\,; \\
\quad F_1 := 1\,; \\
\quad \mathbf{for}\, k\, \mathbf{from}\, 2\, \mathbf{to}\, n\, \mathbf{do}\, F_k := F_{k-1} + F_{k-2}\ \ \mathbf{od}\,; \\
\quad \mathrm{seq}(F_k,\, k = 0..n) \\
\mathbf{end}
\end{array}
$$

```
> Fibonacc(30);
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040

1.4.4 Summation

Example 1: Write a program that adds the squares of the first n odd, positive integers. That is, the sum

$$1^2,\, 2^2,\, 3^2, \cdots, n^2$$

Solution 1: Analysis: The first odd number is 1. If k is an odd number, $k+2$ is the next one

```
# program that compute the sum of the 1st n
#     odd positive integers
# input: n
# output the sum.
#
oddsum1:=proc(n)
   local i, oddnum, s;
   s:=0;                 # initialize the sum as zero
```

```
   oddnum:=1;              # initialize the first odd number
   for i from 1 to n do
      s:=s+oddnum^2;       # accumulating the sum by
                           #   adding the new term
      oddnum:=oddnum+2;    # prepare the next odd number
   od;
   print(`The sum is`, s);
end;
```

Solution 2: Analysis: odd numbers are

$$2k-1, \quad k=1,2,\cdots$$

```
# program that compute the sum of the first n
#     odd positive integers
# input: n
# output the sum.
#
oddsum2:=proc(n)
   local k, s;
   s:=0;                            # initialize the sum as zero
   for k from 1 to n do
      s:=s+(2*k-1)^2;               # accumulate the sum by
                                    # adding the new term
   od;
   print(`The sum is`, s);
end;
```

The programming structure of adding n terms is as follows.

s:=0;
for$i$ from 1 to n do
    {*prepare the new term*}
    s:=s+{*the new term*}
od;

The key is s:=s+{*the new term*}. The variable s (not necessarily s literally) here is used to accumulate the sum. Every time the loop come back to this statement, the value of s adds a new term and the results is assigned to s, replacing its old value. The loop goes from $i=1$ to $i=n$ and s accumulates terms one by one, eventually accumulates all terms.

Example 2. The sine function can be calculated by

$$\sin(x)=\frac{x}{1!}-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\frac{x^9}{9!}-\frac{x^{11}}{11!}+\cdots$$

Write a program, for input values of x and n, to approximate the sine function at x by calculating the sum of the first n terms.

Analysis: Every term is in the form of $\frac{x^k}{k!}$ for odd k , and with alternating +/- sign.

```
# program to approximate sine function
#    by adding the first n terms of Taylor expansion
# input: x,  n,
# output: the approximate value of sin(x)
#
sine:=proc(x,n)
   local i, s, k, sgn;
   s:=0;                             # empty accumulator
   k:=1;                             # first odd number
   sgn:=1;                           # first sign is +
   for i from 1 to n do
      s:=s + sgn*x^k/k!;             # accumulating
      sgn:=-sgn;                     # alternate sign
      k:=k+2;                        # next odd number
   od;
   print(`The sine function`, s);   # output
end;
```

Now test run:

```
>  read(`a:sine.txt`);
```

$sine := \mathbf{proc}(x, n)$
$\mathbf{local}\, i, s, k, sgn;$
$s := 0;$
$k := 1;$
$sgn := 1;$
$\mathbf{for}\, i\, \mathbf{to}\, n\, \mathbf{do}\, s := s + sgn \times x^k \big/ k!;\ sgn := -sgn;\ k := k + 2\, \mathbf{od};$
$\mathrm{print}(`\mathit{The\ sine\ function}`, s)$
$\mathbf{end}$

```
>  x:=evalf(Pi/6);
```

$$x := .5235987758$$

```
>  sine(x,3);
```

$$\mathit{The\ sine\ function}, .5000021328$$

```
>  sine(x,4);
```

$$\mathit{The\ sine\ function}, .4999999921$$

```
>  sine(x,5);
```

$$\mathit{The\ sine\ function}, .5000000003$$

We can compare with the maple built-in function

```
>  sin(x);
```

$$.5000000002$$

The program is as accurate as the build in function sin for $n = 5$.

1.5 Exercises

1. **A mortgage problem**

   Let \$ A be the amount of a mortgage, n the total number of payments, i the interest rate per period of payment. Then the payment \$ R per period is given by the formula:

   $$R = \frac{A\,i}{1 - (1+i)^{(-n)}}$$

   - Write a program that, for input: p, r, y, d as price of the purchase, annual interest rate, number of years, and down payment rate respectively, calculates and prints the monthly payment amount.
   - Use your program to calculate the monthly payment of a \$180,000 house, 20% down, 7.75% annual interest rate for 30 years.
   - Use your program to calculate the monthly payment of a \$15,000 car, 10% down, 9.25% annual interest, rate for 5 years.

   Sample results:

   ```
   > read(`a:mortgage.txt`):
   > price:=180000:  down:=20:  rate:=7.75:  year:=30:
   > mortgage(price,rate,year,down);
   ```

   $$\textit{The monthly payment is}$$
   $$1031.633646$$

2. **Geometry of a circle**

   Write a program that displays radius, circumference, and area of a circle with a given diameter

   Sample results

   ```
   > read(`a:circle.txt`):
   > diameter:=10:
   > circle(diameter);
   ```

   $$\textit{The radius} :, 5.0$$
   $$\textit{The circumference} :, 31.41592654$$
   $$\textit{The area} :, 78.53981635$$

3. **Parallel resistors**

When three electrical resistors with resistance R_1, R_2, and R_3 respectively are arranged in parallel, their combined resistance R is given by the formula

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}$$

Write a program that calculate R

Sample results

```
> read(`a:resistor.txt`):
> R1:=20:  R2:=50:  R3:=100:
> resistor(R1,R2,R3);
```

The combined resistance is

12.50000000

4. **Maximum stress**

The maximum stress in an eccentrically loaded column shown below is given by the formula

$$\sigma_{max} = \frac{P\left(1 + \frac{ec}{r^2 \cos\left(\frac{L\sqrt{\frac{P}{A\,E}}}{2\,r}\right)}\right)}{A}$$

Write a program that calculates and prints the maximum stress. Use the following sample data to test your program

Sample Data

P	A	e	c	R	L	E
4000	29.55	16.6	5.56	4.68	120	3.0×10^7

5. **Card identification**

(See Example 3) Write a program that for input of a number, identify the corresponding card name.

Sample results

```
> read(`a:numcard.txt`):
> numcard(1);
```

ace, heart

```
> numcard(25);
```

queen, spade

```
> numcard(30);
```

4, *diamond*

```
> numcard(0);
```

Error : the input must be 1 – 52

```
> numcard(55);
```

Error : the input must be 1 – 52

```
> numcard(45);
```

6, *club*

6. **Cost calculation**

   A small company offers a car rental plan: \$20.00 a day plus \$0.10/mile for up to 200 miles per day. No additional cost per day for more than 200 miles. (e.g., 3 days with 500 miles cost \$110.00 while 3 days with 800 miles cost \$120.00.) Write a program, for input (1) number of days, and (2) total mileage, calculates the total rental cost.

7. **Cost calculation**

   A travel agency offers a Las Vegas vacation plan. A single traveler costs \$400. If a traveler brings companions, each companion receives a 10% discount. A group of 10 or more receives 15% discount per person. Write a program that, for input number of travelers, print out the total cost.

8. **Real solutions to a quadratic equation**

   For the general quadratic equation

   $$ax^2 + bx + c = 0, \quad a \neq 0$$

   the discriminant Δ

   $$\Delta = b^2 - 4\,ac$$

   - When $\Delta > 0$, there are two real solutions;
   - when $\Delta = 0$, there is only one real solution $-\frac{b}{2\,a}$;
   - there are no real solutions otherwise (i.e. $\Delta < 0$).

   Write a program that for input of a, b, c, print out the number of real solutions and those real solutions themself, if any.

   Sample results

```
> read(`a:quadreal.txt`):
> a:=1:  b:=0:  c:=1:
> quadreal(a,b,c);
```

   There are no real solutions

```
> a:=4:  b:=4:  c:=1:
> quadreal(a,b,c);
```

   There is only one real solution

   $$-.5$$

```
> a:=3:  b:=5:  c:=2:
> quadreal(a,b,c);
```

   There are two real solutions

   $$-.6666666665,\ -1.000000000$$

9. **More on quadratic equations**

   The quadratic equation problem can be extended to more general case: $a = 0$, the equation is reduced to a linear equation and there is only one real solution $-\frac{c}{b}$ while b is nonzero. If $b = 0$ too, the equation is not valid.

Write a program that calculates real solutions of a quadratic equations for all the possible cases.

Sample results

```
> read(`a:quadadv.txt`):
> a:=0:  b:=0:  c:=0:
> quadadv(a,b,c);
```

Error : invalid equation

```
> a:=0:  b:=5:  c:=3:
> quadadv(a,b,c);
```

This is a linear equation with solution

$$\frac{-3}{5}$$

```
> a:=5:  b:=3:  c:=4:
> quadadv(a,b,c);
```

There are no real solutions

```
> a:=1:  b:=2:  c:=1:
> quadadv(a,b,c);
```

There is only one real solution

$$-1.0$$

```
> a:=2:  b:=-9:  c:=6:
> quadadv(a,b,c);
```

There are two real solutions

3.686140663, .8138593385

10. **Tax schedule**

The following is 1996 tax schedule. Write a program to calculate taxes, for input *Status* and *TaxableIncome*

```
   Status 1 (single)

  TaxableIncome          Tax
      0 -  23,350                0.15*TaxableIncome
 23,350 -  56,550      3,502.50+ 0.28*(TaxableIncome- 23,350)
 56,550 - 117,950     12,798.50+ 0.31*(TaxableIncome- 56,550)
117,950 - 256,500     31,832.50+ 0.36*(TaxableIncome-117,950)
256,500 - up          81,710.50+0.396*(TaxableIncome-256,500)

   Status 2 (Married filing jointly or Qualifying Widow(er)

      TaxableIncome          Tax
       0 -  39,000               0.15*TaxableIncome
  39,000 -  94,250     5,850.00+ 0.28*(TaxableIncome- 39,000)
  94,250 - 143,600    21,320.00+ 0.31*(TaxableIncome- 94,250)
 143,600 - 256,500    36,618.50+ 0.36*(TaxableIncome-143,600)
 256,500 - up         77,262.50+0.396*(TaxableIncome-256,500)

   Status 3 (Married filing separately)

      TaxableIncome          Tax
```

```
      0 -  19,500              0.15*TaxableIncome
 19,500 -  47,125     2,925.00+ 0.28*(TaxableIncome- 19,500)
 47,125 -  71,800    10,660.00+ 0.31*(TaxableIncome- 47,125)
 71,800 - 128,250    18,309.25+ 0.36*(TaxableIncome- 71,800)
128,250 - up         38,631.25+ 0.396*(TaxableIncome-128,250)

  Status 4 (Head of household)

     TaxableIncome          Tax
      0 -  31,250          0.15*TaxableIncome
 31,250 -  80,750    4,687.50 + 0.28*(TaxableIncome- 31,250)
 80,750 - 130,800    18,547.50+ 0.31*(TaxableIncome- 80,750)
130,800 - 256,500    34,063.00+ 0.36*(TaxableIncome-130,800)
256,500 - up         79,315.00+ 0.396*(TaxableIncome-256,500)
```

Sample results

```
> read(`a:taxschdl.txt`)
> status:=2:  TaxableIncome:=56890:
> taxschdl(status,TaxableIncome);
```

$$10859.20$$

```
> status:=4:  TaxableIncome:=35280:
> taxschdl(status,TaxableIncome);
```

$$5975.90$$

```
> status:=1:  TaxableIncome:=2590000:
> taxschdl(status,TaxableIncome);
```

$$.1005776500\ 10^7$$

11. **Cubic root of 21**

The sequence

$$x_k = \frac{20\, x_{k-1} + 21\, (\frac{1}{x_{k-1}})^2}{21}, \quad x_1 = 1, \quad k = 2, 3, \cdots$$

converges to the cubic root of 21 slowly. Write a program to print out the first n terms of the sequence and observe how many steps does it take to reach its limit within 5 digits.

12. $\frac{\pi}{4}$

Write a program to calculate

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

and verify that it equals to $\frac{\pi}{4}$.

13. **Combination of Lucas and Fibonacci sequences**

The Lucas sequence is defined by

$$L_0 = 2,\ L_1 = 1,\ L_k = L_{k-1} + L_{k-2}, \quad k = 2, 3, \cdots, n$$

Write a program that print out the first n terms of

$$L_k{}^2 - 5\,F_k{}^2,\ k = 1, 2, \cdots, n$$

where F_k is the k-th term of the Fibonacci sequence. Make a conjecture about its values. Can you prove your conjecture?

14. **The cosine function**

    The cosine function can be calculated by

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \frac{x^{12}}{12!} - \cdots$$

    Write a program to approximate the sine function by calculating the sum of the first n terms. How many terms is it necessary to be as accurate as the build in function for $x = \frac{\pi}{3}$

15. **The sum of the Fibonacci sequence**

    Write a program to compute the sum of the first n terms of Fibonacci sequence.

1.6 Further reading material

1.6.1 Documentation of programs

Programs should be well documented for reading, revision and applications. In a Maple program, a "#" sign starts a comment in any line. A programmer should, at least, include comments in the program to tell users and other programmers the following information

1. the purpose of the program
2. the required input
3. the output

It is also a good idea to comment all important steps inside the program.

The use of indentation and empty lines can also improve the readability of programs. Compare the following two programs:

```
oddsum:=proc(n)
local i, oddnum, s;
s:=0;
oddnum:=1;
for i from 1 to n do
s:=s+oddnum^2;
oddnum:=oddnum+2;
od;
print(`The sum is`, s);
end;
```

The one below is functionally the same program, but it is much easier to read with proper indentation and line separation.

```
#
# Program that calculate the squares
# of 1st n odd integers
#    input: n --- integer
#
oddsum:=proc(n)
   local i, oddnum, s;
```

```
   s:=0;                         # prepare initial values
   oddnum:=1;                    # for the summation

   for i from 1 to n do          # the loop of summation
      s:=s+oddnum^2;
      oddnum:=oddnum+2;
   od;

   print(`The sum is`, s);       # output the sum

end;
```

The blank lines separate programs into three blocks based on their functionalities. By indenting the statements inside the loop, the repeating statements are obvious.

A programmer, when working alone, can use his/her own programming style. In reality, programers are most likely working in groups. Therefore it is important to have a common source code writing style. We suggest the following guidelines in Maple programming:

1. A blank line between logical blocks of statements.
2. Align the opening comments, the program definition line with the "end" statement, and have a three-space indentation for every line between these two lines.
3. If an if–block is used, align "if", "elif", "else", "fi" statements, and have an additional three-space indentation for every line between them.
4. If a loop is used, align "for" and "od" statements, and have an additional three-space indentation for every line between them.

Programming work will be a lot easier if those guidelines are followed. It is also important that you follow these guidelines WHILE writing your program, instead of after the program is done.

1.6.2 Formated printing

Maple's formated printing is quite similar to that in C programming language. The command syntax is that

printf(*'format'*, *expression_sequence*);

where *format* specifies how Maple is to write the *expression_sequence*. For example "%d" is to tell Maple to write the expression as a signed decimal integer:

```
> k:=-23456; j:=65432;
```

$$k := -23456$$
$$j := 65432$$

```
> printf(`The integer %d is assigned to variable k`,k);

The integer -23456 is assigned to variable k

> printf(`The integer %d plus integer %d is %d`,k,j,k+j);

The integer -23456 plus integer 65432 is 41976
```

The *format* part of printf must be inside the quotes ` `. The "%" begins the format specification. The simplist way of specifying the printing format is

%code

where code can be one of the following (copied from Maple help file)

```
d       ---- The object is formatted as a signed decimal integer.
o       ---- The object is formatted as an unsigned octal (base 8)
             integer.


x or X  ---- The object is formatted as an unsigned hexadecimal (base 16)
             integer. The digits corresponding to the decimal numbers 10
             through 15 are represented by the letters A through F if X
             is used, or a through f if x is used.

e or E  ---- The object is formatted as a floating point number in
             scientific notation. One digit will appear before the
             decimal point, and the number of digits specified by the
             precision will appear after the decimal point (6 digits
             if no precision is specified).  This is followed by the
             letter e or E, and a signed integer specifying a power of
             10. The power of 10 will have a sign and at least 3 digits,
             with leading zeroes added if necessary.

f       ---- The object is formatted as a fixed point number. The number
             of digits specified by the precision will appear after the
             decimal point.

g or G  ---- The object is formatted using e (or E if G was specified),
```

```
              or f format, depending on its value. If the formatted value
              would contain no decimal point, d format is used. If the
              value is less than 10^(-4) or greater than 10^precision,
              e (or E) format is used.  Otherwise, f format is used.

c       ---- The object, which must be a Maple string containing exactly
              one character, is output as a single character.

s       ---- The object, which must be a Maple string, is output as a
              string of at least width characters (if specified) and at
              most precision characters (if specified).

a       ---- The object, which can be any Maple object, is output in
              correct Maple syntax. At least width characters are output
              (if specified), and at most precision characters are output
              (if specified).  NOTE:  truncating a Maple expression by
              specifying a precision can result in an incomplete or
              incorrect Maple expression in the output.

m       ---- The object, which can be any Maple object, is output in
              Maple's ".m" file format. At least width characters are
              output (if specified), and at most precision characters are
              output (if specified).  NOTE:  truncating a Maple ".m"
              format expression by specifying a precision can result in
              an incomplete or incorrect Maple expression in the output.

M or N  ---- The object, which can be any Maple object, is output in
              OpenMath syntax. At least width characters are output
              (if specified), and at most precision characters are output
              (if specified).  NOTE: truncating an OpenMath expression by
              specifying a precision can result in an incomplete or
              incorrect Maple expression in the output.  ALSO NOTE: The
              outut uses OpenMath syntax, but assumes Maple semantics;
              only Maple will be able to process the result. To produce
              an OpenMath compliant result, additional preprocessing is
              required.  If the "\%N" format was specified, no
              back-references will appear in the output.

\%      ---- A percent symbol is output verbatim.
```

Examples:

```
> s:=4.0/3;
```

$$s := 1.333333333$$

```
> printf(`Real number %e is printed in scientific notation`,s);
```

```
Real number 1.333333e+00 is printed in scientific notation
```

```
> printf(`To be more specific, %15.4e is printed \nusing a total of
> 15 characters (including spaces) \nwith 4 digits after decimal
> point`,s);
To be more specific,      1.3333e+00 is printed
using a total of 15 characters (including spaces)
with 4 digits after decimal point
> printf(`%8.3f can be printed as a floating point number too.\nIt
> occupies 8 characters with 3 digits after decimal point.`,s);
   1.333 can be printed as a floating point number too.
It occupies 8 characters with 3 digits after decimal point.
```

In examples above, "\n" is the specification for changing line.

Chapter 2

Iterations

2.1 Methods of golden section and bisection

The method of golden section can be applied to solve the optimization problem of unimodel functions. It is similar to the bisection method for finding zeros of a function. We'll use the method of golden section as an example to show the programming techniques and leave bisection method as a project for students.

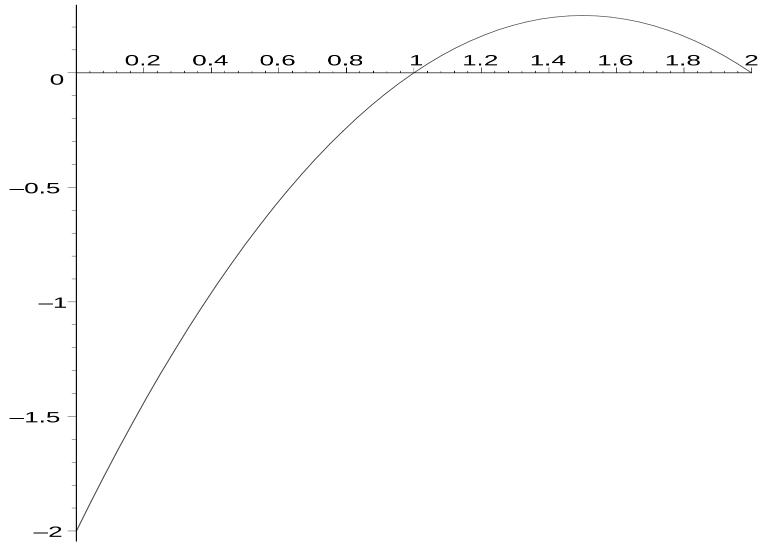
2.1.1 Unimodel functions

A function is called unimodel if it is strictly increasing, reaches its maximum, and then strictly decreasing. A typical unimodel function, with graph generated with Maple commands:

```
> f:=x->-x^2+3*x-2;
```

$$f := x \to -x^2 + 3\,x - 2$$

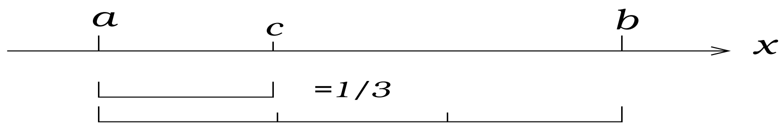
```
> plot(f,0..2);
```



The maximum value of the above function occurs at $x = 1.5$.

2.1.2 Golden section

Let $[a, b]$ be an interval with a point c inside. The location of c in $[a, b]$ can be described as "how much is c to the right of a". For example, we may say c is one third to the right of a, meaning the length of $[a, c]$ is $\frac{1}{3}$ of that of $[a, b]$. This "one third" is the *section ratio* of c in $[a, b]$.



In this case, the value of c equals that of a plus the length of $[a, c]$, which is $\frac{1}{3}$ of the length of $[a, b]$. That is,

$$\begin{aligned} c &= a + (c - a) \\ &= a + \frac{1}{3}(b - a) \\ &= (1 - \frac{1}{3})a + \frac{1}{3}b \end{aligned}$$

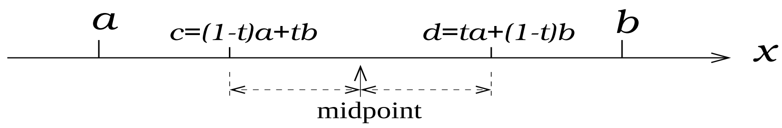
For the same reason, if we know that a point $d \in [a, b]$ with a section ratio equals to t where $0 \le t \le 1$, then

$$d = (1-t)\,a + t\,b$$

and the section ratio

$$t = \frac{c-a}{b-a} = \frac{\text{length of } [a,c]}{\text{length of } [a,b]}$$

Let c be a point in the interval $[a,b]$ with section ratio t. I.e. $c = (1-t)a + tb$. If we reverse the role of t and $1-t$, we have another point $d = ta + (1-t)b$, which is symmetric to c about the midpoint of $[a,b]$. These pair of points are said to be conjuagate to each other in $[a,b]$

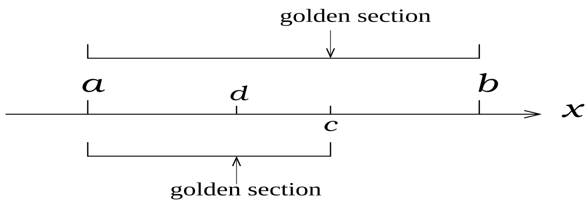


There is a special section ratio τ, called *golden section ratio*. It is special because if c cuts the interval $[a,b]$ with this section ratio τ,

$$c = (1-\tau)a + \tau b$$

then its conjuagate point d cuts the interval $[a,c]$ with the same golden section ratio

$$d = \tau a + (1-\tau)b = (1-\tau)a + \tau c$$



It can be verified that the golden section ratio is

$$\tau = \frac{-1+\sqrt{5}}{2}$$

which is approximately 0.618.

2.1.3 The method of golden section

Let f(x) be a unimodel function on $[a,b]$. We know there is a maximum point x_* of f(x) inside the interval $[a,b]$. Our objective is to shrink the interval so that it still contains x_*, thereby we can zeroin on x_*.

As shown in Figure 2.1, let m_l and m_r be a pair of conjuage section point corresponding to the golden section ratio, we may call them the mid-left and mid-right golden section points of $[a, b]$. We shall shrink $[a, b]$ to either left subinterval $[a, m_r]$ or the right subinterval $[m_l, b]$. We are looking for the unique maximum point of the unimodel function f(x). So we compare the values of $f(m_l)$ and $f(m_r)$. If $f(m_r) > f(m_l)$ as illustrated in Figure 2.1, then the interval $[m_l, b]$ containing m_r is the interval of choice. Similarly, if $f(m_l) > f(m_r)$, we would shrink $[a, b]$ to $[a, m_r]$.

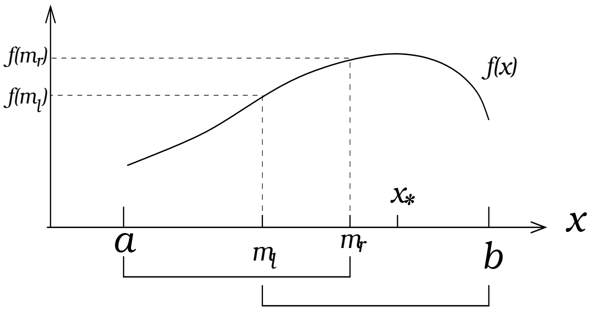


Figure 2.1: The method of golden section: interval $[m_l, b]$ is chosen over $[a, m_r]$ because $f(m_r) > f(m_l)$

We therefore have a simple rule of choosing subintervals:

- *If the "left value" $f(m_l)$ is larger, choose the "left subinterval" $[a, m_r]$.*
- *If the "right value" $f(m_r)$ is bigger, choose the "right subinterval" $[m_l, b]$*

The beauty of the method is that each subinterval is cut by either m_l or m_r with golden section ratio. Therefore, to shrink the subinterval, we need only the conjuagate section point and only one additional function evaluation.

So the method of golden section can be described in the following pseudo-code:

Input: a, b, f and error tolerance δ

Make sure *delta* is positive

Start with working interval $[\alpha, \beta] = [a, b]$

Set τ the golden section ratio

Find the pair of golden section points m_l and m_r

Calculate $v_l = f(m_l)$ and $v_r = f(m_r)$

While $|\beta - \alpha| > \delta$, repeat the following as a loop

If $v_l > v_r$ then

Replace $[\alpha, \beta]$ with $[\alpha, m_r]$

Replace m_r with m_l

Replace v_r with v_l

Replace m_l with $(1-\tau)\alpha + \tau\beta$

Calculate $v_l = f(m_l)$

else

Replace $[\alpha, \beta]$ with $[m_l, \beta]$

Replace m_l with m_r

Replace v_l with v_r

Replace m_r with $\tau\alpha + (1-\tau)\beta$

Calculate $v_r = f(m_r)$

end if

The actual program based on the pseudo-code:

```
# The program that calculate the maximum point of a unimodel function
# input: f    --- the unimodel function, define in Maple as operator,
#                 e.g. f:=x->-x^2+3*x-2;
#       a, b --- end points of the interval [a,b]
#       tol  --- the error tolerance
#
goldsec:=proc(f,a,b,delta)
   local goldratio, gratio, alpha, beta, ml, mr, fml, fmr, stepcount;

   if delta <= 0.0 then                    # avoid negative delta
      print(`error tolerance must be positive`);
      RETURN();
   fi;

   tau := evalf( 0.5*(-1.0+sqrt(5.0)) );   # golden section ratio
   alpha:=a; beta:=b;                      # working interval
   ml:=(1-tau)*alpha+tau*beta;             # the mid-left point
   mr:=tau*alpha+(1-tau)*beta;             # the mid-right point
   vl:=f(ml);  vr:=f(mr);                  # function values

   while abs(beta-alpha) >= delta do       # the main loop

      if vl > vr then
         beta:=mr;                         # undate working interval
         mr:=ml;  vr:=vl;                  # update mr and vr
         ml:=(1-tau)*alpha+tau*beta;       # the new mid-left
         vl:=f(ml);                        # the new vl
      else
         alpha:=ml;                        # update working interval
         ml:=mr;  vl:=vr;                  # update mr and vr
```

```
            mr:=tau*alpha+(1-tau)*beta;        # the new mid-right
            vr:=f(mr)                          # the new vr
          fi;

      od;

      print(`The solution is`);
      print( 0.5*(aa+bb) );                    # output

    end;
```

In the program, we used command

$$\text{RETRUN}()$$

which force the machine exit the program.

Set the number of digits to be 20:

```
> Digits:=20;
```

$$Digits := 20$$

```
> read(`a:goldsec.txt`):
> a:=0; b:=2; tol:=0.001;
```

$$a := 0$$
$$b := 2$$
$$tol := .001$$

```
> goldsec(f,a,b,tol);
```

The solution is

1.5001934984462164631

If we reduce the tolerance to 0.0002, we have

```
> goldsec(f,a,b,0.00002);
```

The solution is

1.4999951775621607752

```
> goldsec(f,a,b,0.000000001);;
```

The solution is

1.4999999999534881866

2.1.4 The while–do loop

The implementation of golden section method involve another type of loops:

while *condition* do
 block of statements
od;

In this loop, the *block of statements* will be repeated as long as the *condition* remains "true" Using the method of golden section as an example, we started with a working interval $[\alpha, \beta]$ and enter the while–do loop. If the interval is small enough in the first place (i.e. $\beta - \alpha \leq \delta$), all the statements inside the loop(i.e. between "do" and "od") will be ignored and "print" statements will follow. Otherwise, the machine will enter the loop shrink the working interval, reach "od" and <u>come back</u> to "while" to recheck the condition $\beta - \alpha \geq \delta$. Eventually the *condition* will become "false" the machine jumps to the line following "od".

There is a danger of using while–do loops. It may run forever if the condition can never be met. For example, the program goldsec, with accidental input of negative error tolerance, would be a dead loop if there were no statements

```
if delta <= 0.0 then                    # avoid negative delta
   print(`error tolerance must be positive`);
   RETURN();
fi;
```

2.1.5 The bisection method

From the Intermediate Value Theorem in Calculus, let f(x) be a continous function on the interval $[a, b]$ such that f(a) and f(b) have different signs, then there exists a number $x_* \in [a, b]$ such that $\mathrm{f}(x_*) = 0$.

To find x_*, we can cut the interval with the mid-point $mp = \frac{a+b}{2}$ and calculate f(mp). According to the sign of f(mp),

$$\text{the solution } x_* \text{ is in or equal to } \begin{cases} [a,\ mp] & \mathrm{f}(a)\,\mathrm{f}(mp) \leq 0 \\ mp & \mathrm{f}(mp) = 0 \\ [mp,\ b] & \textit{otherwise} \end{cases}$$

We can thereby replace the interval $[a,\ b]$ with the new subinterval, either $[a,\ mp]$ or $[mp,\ b]$This process, which is called ***the bisection method*** can be repeated until the length of the final interval is of length less than the error tolerance. The midpoint of the final interval can be then output as the approximate solution.

The implementation of the bisection method is similar to the method of golden section, and left to readers.

2.2 Newton's and other iterative method

2.2.1 Newton's iteration

For a given function $\mathrm{f}(x)$ which is differentiable, the iteration

$$x_k = x_{k-1} - \frac{\mathrm{f}(x_{k-1})}{\mathrm{f}'(x_{k-1})}, \quad k = 0,\, 1,\, 2,\, \cdots$$

is called Newton's iteration. If the starting point x_0 is near a zero x_* of the function f, the iteration generates a sequence of points $x_1, x_2, \cdots$ that converges to the solution x_*. Usually Newton's iteration converges very fast, if it converges at all.

For example, let $\mathrm{f}(x) = x - \cos(\pi\, x)$, there is a solution 0.3769670099. If we start at $x_0 = .5$

```
> x:=array(0..10);
```

$$x := \mathrm{array}(0..10,\, [])$$

```
> f:=x->x-cos(Pi*x);
```

$$f := x \to x - \cos(\pi\, x)$$

```
> g:=D(f);  #find the derivative of f
```

$$g := x \to 1 + \sin(\pi\, x)\, \pi$$

```
> x[0]:=0.5;
```

$$x_0 := .5$$

```
> x[1]:=evalf(x[0]-f(x[0])/g(x[0]));
```

$$x_1 := .3792734965$$

Two digits are correct after one step.

```
> x[2]:=evalf(x[1]-f(x[1])/g(x[1]));
```

$$x_2 := .3769695051$$

Five digits are correct after two steps.

```
> x[3]:=evalf(x[2]-f(x[2])/g(x[2]));
```

$$x_3 := .3769670094$$

In only three steps we got the solution up to 9 digits.

The above process can be implemented using a loop.

```
> f:=x->x-cos(Pi*x);
```

```
g:=D(f);
for k from 1 to 10 do
   delta:=evalf(f(x[k-1])/g(x[k-1])):
   x[k]:= x[k-1] - delta:
   printf( " x[\%2d]= \%15.10f   delta=\%15.10f \\n",
           k, x[k], delta):
od:
```

```
x[ 1]=     .3792734965   delta=    .1207265035

x[ 2]=     .3769695051   delta=    .0023039914

x[ 3]=     .3769670094   delta=    .0000024957

x[ 4]=     .3769670092   delta=    .0000000002

x[ 5]=     .3769670093   delta=   -.0000000001

x[ 6]=     .3769670094   delta=   -.0000000001

x[ 7]=     .3769670092   delta=    .0000000002

x[ 8]=     .3769670093   delta=   -.0000000001

x[ 9]=     .3769670094   delta=   -.0000000001

x[10]=     .3769670092   delta=    .0000000002
```

There are several important issues about Newton's iteration:

- Newton's iteration converges *locally*. Namely, it could fail, if the starting iterate x_0 is not close enough to the solution x_s.
- Even if Newton's iteration does converge, we generally don't know in advance how many steps are needed to reach the desired accuracy.

Therefore, we must set certain *stop criteria* to terminate Newton's iteration.

First, there must be a limit to the number of steps for Newton's iteration. An input integer n is needed so that Newton's iteration stops at the step x_n. Secondly, the iteration should stop when the accuracy is good enough. The example above shows that the magnitude of

$$\delta_k = \frac{\mathrm{f}(x_{k-1})}{\mathrm{f}'(x_{k-1})}$$

indicates the accuracy of x_k. Actually, for a given tolerance *tol*,

$$|\delta_k| \leq tol$$

a widely used criterion for stoping Newton's iteration.

From the above discussion, we can set up a loop of iteration that stops either at step n or when $|\delta_k| \leq tol$, which ever achieves first. This is can be done with the following Maple loop command that combines for–do and while–do:

for *loop_index* from $\cdots$ to $\cdots$ while *condition* do
  statements to be repeated
od;

Using the previous example

```
> delta:=1.0:
  for k from 1 to 10 while abs(delta)>10.0^(-8) do
     delta:=evalf(f(x[k-1])/g(x[k-1])):
     x[k]:= x[k-1] - delta:
     printf( " x[\%2d]= \%15.10f   delta=\%15.10f \\n",
             k, x[k], delta):
  od:
```

```
x[ 1]=     .3792734965   delta=    .1207265035

x[ 2]=     .3769695051   delta=    .0023039914

x[ 3]=     .3769670094   delta=    .0000024957

x[ 4]=     .3769670092   delta=    .0000000002
```

This loop stops at the fourth step because $\delta > 10^{(-8)}$ is no longer true. In worst case, the loop will stop at 10-th step.

A Newton's iteration program can be implemented with the following pseudo-code:

Input: function f, initial iterate *x0*, step limit n, tolerance *tol*

calculate derivative of f and assign it to g

generate an (empty) array x with index from 0 to n

assign the value *x0*

set an initial value delta larger than tol

use "for...from...to...while...do" loop to carry out the iteration check if abs(delta)<tol.

if so, print the last iterate

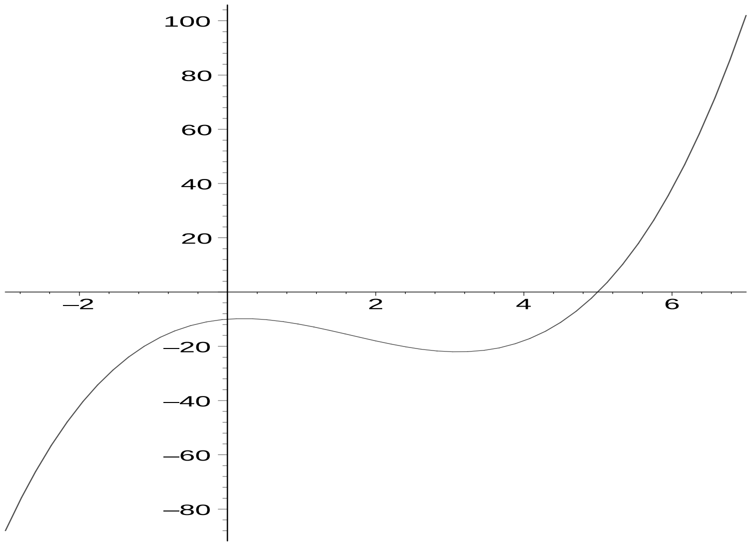
otherwise, print out a failure message.

The actual implementation will be left as an exercise. Here are some examples of Newton's iteration.

```
> f:=x->x^3-5*x^2+2*x-10;
```

$$f := x \to x^3 - 5\,x^2 + 2\,x - 10$$

```
> plot(f,-3..7);
```



The graph of f(x) shows that there is a zero around $x = 5$. Now lets apply Newton's iteration

```
> x0:=4.0: tol:=10.0^(-8): n:=10:
> read(`a:newton.txt`):
> newton(f,x0,n,tol);
```

```
x[ 1]=    5.8000000000   delta=  -1.8000000000

x[ 2]=    5.1652715940   delta=    .6347284061

x[ 3]=    5.0092859510   delta=    .1559856433

x[ 4]=    5.0000317760   delta=    .0092541752
```

```
 x[ 5]=    5.0000000010   delta=    .0000317752

 x[ 6]=    5.0000000010   delta=   0.0000000000

Newton's iteration successful in  6 steps

The solution is         5.0000000010
```

Starting from certain points you'll see that Newton's iteration fails

```
> y0:=-3.0:
> newton(f,y0,n,tol);
```

```
 x[ 1]=   -1.5084745760   delta=  -1.4915254240

 x[ 2]=    -.3447138130   delta=  -1.1637607630

 x[ 3]=    1.6065726800   delta=  -1.9512864930

 x[ 4]=    -.8521934710   delta=   2.4587661510

 x[ 5]=     .4039993150   delta=  -1.2561927860

 x[ 6]=   -6.0088494770   delta=   6.4128487920

 x[ 7]=   -3.5470641770   delta=  -2.4617853000

 x[ 8]=   -1.8900892540   delta=  -1.6569749230

 x[ 9]=    -.6757701190   delta=  -1.2143191350

 x[10]=     .7009957250   delta=  -1.3767658440

 Newton's iteration unsuccessful in 10 steps
```

2.3 Exercises

1. **The ratio of golden section**

   Show that the ratio of golden section is $\frac{-1+\sqrt{5}}{2}$

2. **Programming the bisection method**

   Using the program for the method of golden section as an example, write a program for solving the equation

   $$\mathrm{f}(x) = 0$$

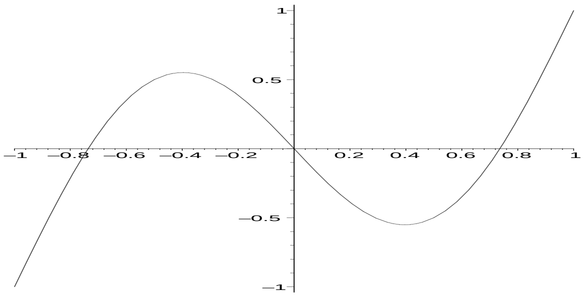
   using the bisection method.

   Sample results We are interested in the zeros of the function $\mathrm{f}(x) = x - \sin(\pi\, x)$ in the interval $[-1, 1]$.

```
> f:=x->x-sin(Pi*x);
```

$$f := x \to x - \sin(\pi\, x)$$

```
> plot(f,-1..1);
```



There are three zeros in intervals $[-1,-0.5]$, $[-0.5,0.5]$, and $[0.5,1]$. Most importantly, the function f(x) have opposite signs at the end points of each interval. Therefore we can apply the bisection method to locate them.

```
> read(`a:bisect.txt`):
> bisect(f,-1,-0.5,0.00000001);
```

The approximate solution is

$$-.7364844497$$

```
> bisect(f,-0.4,0.5,0.00000001);
```

The approximate solution is

$$-.372532403\;10^{-9}$$

```
> bisect(f,0.5,1,0.00000001);
```

The approximate solution is

$$.7364844497$$

Another example: suppose we are interested in the equation

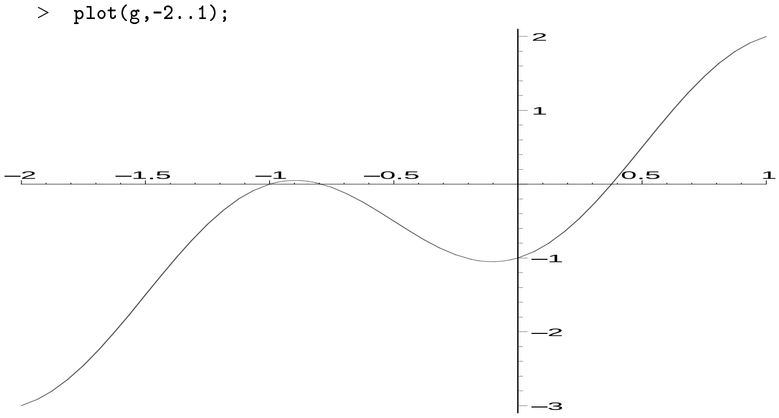
$$x = \cos(\pi\, x)$$

on the interval $[-2,1]$. We investigate the function by plotting the graph:

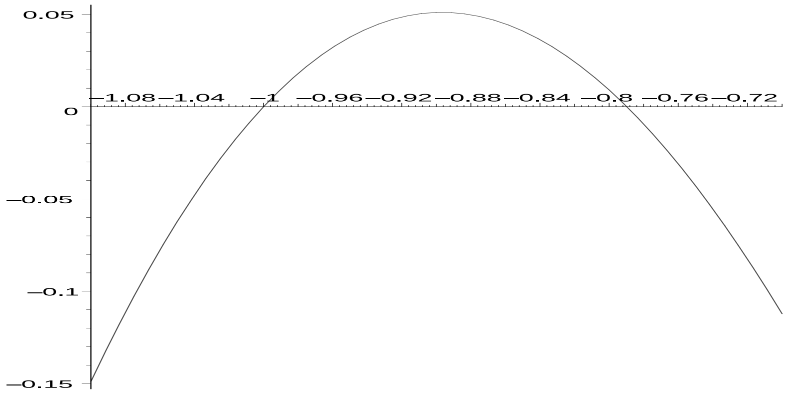
```
> g:=x->x-cos(Pi*x);
```

$$g := x \to x - \cos(\pi\, x)$$

```
> plot(g,-2..1);
```




```
> plot(g,-1.1..-0.7);
```



Therefore there are three solutions.

```
> bisect(g,-1.1,-0.9,0.00000001);
```

The approximate solution is

-1.000000003

```
> bisect(g,-0.9,-0.7,0.00000001);
```

The approximate solution is

$-.7898326312$

```
> bisect(g,0,1,0.000000001);
```

The approximate solution is

$.3769670099$

Try $\mathrm{h}(x) = \sin(x) - e^{(-x)}$ on the interval $[0, 7]$.

3. **Programming Newton's iteration**

   Write a Newton's iteration program, according to the pseudo-code in Section 2.2.1 and following the examples there.

4. **The Euclidean Algorithm**

   A common divisor of two integers m and n is an integer d that divides both m and n. For example, integer 60 and 45 have common divisors 3, 5, and 15. In fact, 60=(2)(2)(3)(5) and 45=(3)(3)(5). The largest number among the common divisors m and n is called the greatest common divisor of m

$$gcd(60, 45) = 15$$

   The classical method of finding the greatest common divisor is the Euclidean algorithm:

   Suppose $n < m$.

   (1) Let m be divided by n and let r be the remainder.

   (2) If $r = 0$, then n is the gcd, exit the process.

   (3) Otherwise, replace the pair (m, n) with the pair (n, r), and go back to (1).

For example, we want to find $gcd(13578, 9198)$:

```
> m:=13578; n:=9198;
```

$$m := 13578$$
$$n := 9198$$

```
> r:=m mod n;
```

$$r := 4380$$

```
> m:=n; n:=r; r:=m mod n;
```

$$m := 9198$$
$$n := 4380$$
$$r := 438$$

```
> m:=n; n:=r; r:=m mod n;
```

$$m := 4380$$
$$n := 438$$
$$r := 0$$

Now the gcd is 438 because the remainder $r = 0$.

Write a program that, for any positive integer input m and n, prints out gcd(m,n) using the Euclidean algorithm.

Sample results

```
> read(`a:gcdiv.txt`):
> gcdiv(13578,9198);
```

The greatest common divisor is

438

```
> gcdiv(9198,13578);
```

The greatest common divisor is

438

```
> gcdiv(60,45);
```

The greatest common divisor is

15

5. **Mortgage interest**

   Let the *amount of a mortgage* be $\$A$, with *interest rate per payment period* be i, and the total *number of payments* be n, then the *payment* amount $\$PMT$ *per period* can be calculated with the following formula:

   $$PMT = A \frac{i}{1 - (1+i)^{(-n)}}$$

   Suppose you want to buy a \$120,000 house with 20% down. You can afford \$713/month payment for 30 years. What is the interest rate you need to shop for?

6. **Mortage points**

   Banks often charges points for lending a mortgage. One point means 1% of the mortgage amount. Usually you can borrow that additional amount into that mortgage. Let p be the number of points the bank charges. Your actual amount of mortgage increases from $\$A$ to $\$A(1+0.01p)$. Thus the periodic payments increases to

$$PMT = A\,(1+.01\,p)\,\frac{i}{1-(1+i)^{(-n)}}$$

Banks offer different rate and point combinations and make consumers confused. However, every rate/point combination is equivalent to an *actual interest rate* with 0 point. It makes sense (and cents) to compare different rate/point combination via comparing their actual rates. Find the actual annual rate for a 30 year mortgage at annual rate of 7.75% with 3 points.

(Hint: 1. You may assume an amount of mortgate, say $100,000. The final answer should be independent of that amount. 2. The crucial idea: there is a payment calculated by the formula above for the rate/point combination. With a monthly interest rate x with no point, there is also a would-be payment. If that x is equivalent to the rate/point combination, those two payments must be the same.)

7. **More on mortgage points**

   Banks often charges points for lending a mortgage. One point means 1% of the mortgage amount. Usually you can borrow that additional amount into that mortgage.

   Let p be the number of points the bank charges. Your actual amount of mortgage increases from $\$A$ to $\$A(1+0.01p)$. Thus the periodic payments increases to

$$Pmt = \frac{A\,(1+.01\,p)\,i}{1-(1+i)^{(-n)}}$$

   Banks offer different rate and point combinations and make consumers confused. However, every rate/point combination is equivalent to an *actual interest rate* with no point. It makes sense (and cents) to compare different rate/point combination via comparing their actual rates. Write a program that, for input

```
Amount      --- amount of mortgage

annual_rate --- the percentage of annual rate
                (i.e.  annual_rate=7.75 for 7.75%)

points         --- the number of points

years          --- the number of years
```

print out the actual annual rate with zero point. For \$150000 mortgage of 30 years, use your program to calculate the actual annual rate for (i) 7.75%, 3 points, (ii) 8%, 1 point.

Hint:

(a) Let x be any monthly rate with zero point, the corrsponding payment is $\frac{A\,x}{1-(1+x)^{(-n)}}$. To make it equivalent to the payment calculated with points, you have to solve the equation

$$\frac{A\,x}{1-(1+x)^{(-n)}} = \frac{A\,(1+.01\,p)\,i}{1-(1+i)^{(-n)}}$$

(b) Define a function $f(x)$ such that $f(x) = 0$ equivalent to the equation above

(c) Make careful analysis to find the interval $[a, b]$ such that $f(a)$ and $f(b)$ have different signs. It is good idea to know the meaning of $f(x) > 0$ and $f(x) < 0$.

(d) Fact 1: Adding points makes payment higher. So monthly rate i with 0 point has **lower** payment than the rate i with any point.

(e) Fact 2: In practical cases, points don't double the actual rate. That is, monthly rate $2i$ with 0 point has **higher** monthly payment than monthly rate i with points.

8. **The best rational approximation**

Let r be a real number, such as $\sqrt{2}$, π, $e$to approximate it. For example $\sqrt{2}\frac{3}{2}$, $\frac{7}{5}$, $\cdots$. One can get better approximation if larger denomenator is allowed. The objective is: for a given bound N on the denomenator, what is the best rational approximation to r.

There is a simple method to find the best rational approximation $\frac{p}{q}$. Start from $p = 0$, and $q = 1$. The initial error is $\left|\frac{p}{q} - r\right| = |r|$. Repeat the following until either $N < q$

- If $\frac{p}{q} < r$, increase p by 1
- Otherwise, increase q by 1, to obtain a new set of p, q
- With this new set of p and q, calculate the current error $\left|\frac{p}{q} - r\right|$.
- If this error is less than the previous error, a better approximation is obtained. Update the the error and print out this set of p and q.

Write a program that carry out the above method for input r and N. For a given N, the above method output a sequence of ever better approximation up to the best approximation within the denomenator limit N.

Use your program to print out sequences of every better approximation to $\sqrt{2}$, π, and e, with N to tens of thousands.

Chapter 3

Arrays

3.1 Arrays

3.1.1 The use of arrays

An array is used to store a sequence of data. For example, if we want to store the first 11 terms of Fibonacci sequence, F_0, F_1, $\cdots$ F_{10} we first open an (empty) array with index range from 0 to 10.

```
> fib:=array(0..10); # define an (empty) array
```

$$fib := \mathrm{array}(0..10, [])$$

Then assign the values to each entry of the array:

```
> fib[0]:=0; fib[1]:=1; # initialize the Fibonacci sequence
```

$$fib_0 := 0$$
$$fib_1 := 1$$

```
> for i from 2 to 10 do           # generate remaining

> fib[i]:=fib[i-1]+fib[i-2];  # entries

> od;
```

$$fib_2 := 1$$
$$fib_3 := 2$$
$$fib_4 := 3$$
$$fib_5 := 5$$
$$fib_6 := 8$$

$$fib_7 := 13$$
$$fib_8 := 21$$
$$fib_9 := 34$$
$$fib_{10} := 55$$

Generally, an array is defined by

$$>\{array\ name\}\text{:=array(}\{leading\ index\}..\{ending\ index\}\text{);}$$

Each entry of the array is referenced as $\{arrayname\}[\{index\}]$. You may use ?array in Maple to open documentation about arrays.

```
> fib[6];
```

$$8$$

```
> fib[10];
```

$$55$$

3.1.2 The command "seq"

The command "seq" is very useful in generating a sequence with a known pattern:

for example, the partial Fibonacci sequence f_0, f_1, $\cdots$, f_{20} is a sequence with pattern

$$f_i, i = 0, 1, \cdots, 10$$

In earlier calculation, this sequence was generated and stored as

fib[1], fib[2], ..., fib[10].

Namely, fib[i] for i from 0 to 10:

```
> seq( fib[i], i=0..10 );
```

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55$$

In general, seq(f,i=m..n) generate a sequence in ther form of f for i from m to n. For example, to generate a sequence of odd numbers:

```
> seq( 2*i-1, i=1..20 );
```

$$1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39$$

To generate a sequence of perfect squares:

```
> seq( i^2,  i=1..15 );
```

1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225

3.1.3 Initialize an array using seq

An array can be initialize directly

```
> t:=array(1..5,[2,6,8,10,s]);
```

$$t := [2, 6, 8, 10, s]$$

We may initialize an array with command seq, if the entries of the array have a clear pattern. For example: To generate an array of even numbers:

```
> even_number:=array(1..20, [seq(2*i, i=1..20)]);
```

$$even\_number := [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40]$$

After the array is so generated, we can conduct computation with the array. For example, the sum of the 8-th and 12-th even numbers:

```
> even_number[8]+even_number[12];
```

$$40$$

Example: The array of perfect squares:

```
> squ:=array(1..50,[seq(j^2, j=1..50)]);
```

$squ := [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361,$
$400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961, 1024, 1089, 1156,$
$1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849, 1936, 2025, 2116, 2209,$
$2304, 2401, 2500]$

3.1.4 Entries of an array

You can define any entry of an existing array as any type of data allowed in Maple, even as another array.

```
> s:=array(1..3);
```

$$s := \mathrm{array}(1..3, [])$$

```
> s[1]:=5; s[2]:=`John Doe`;  s[3]:=array(1..3,[Mark,Bob,Jack]);
```

$$s_1 := 5$$
$$s_2 := John\ Doe$$
$$s_3 := [Mark, Bob, Jack]$$

Since s[3] is an array, to reference its 2nd entry:

```
> s[3][2];
```

$$Bob$$

More examples:

Define an array of four Names, each name consists of first and last names. At beginning, open an array of 4 entries, each entry is an (empty) array of 2 entries:

```
> Name:=array(1..4,[seq( array(1..2), i=1..4)]);
```

$$Name := [\%1, \%1, \%1, \%1]$$
$$\%1 := [?_1, ?_2]$$

```
> Name[1][1]:=Bill:  Name[1][2]:=Clinton:
> Name[2][1]:=Al:  Name[2][2]:=Gore:
> Name[3][1]:=Bob:  Name[3][2]:=Dole:
> Name[4][1]:=Jack:  Name[4][2]:=Kemp:
> eval(Name);
```

$$[[Bill, Clinton], [Al, Gore], [Bob, Dole], [Jack, Kemp]]$$

The third name is Bob Dole:

```
> eval(Name[3]);
```

$$[Bob, Dole]$$

The first name of the fourth person is Jack:

```
> Name[4][1];
```

$$Jack$$

3.1.5 Example: An array of prime numbers

A prime number is a number that can only be divided by 1 and itself. Maple has a function "isprime" to identify prime numbers. For example, we know that 3 is prime but 4 is not:

```
> isprime(3);
```

$$true$$

```
> isprime(4);
```

$$false$$

We can write a simple program to put n prime numbers in an array:

```
# The program of generating an array of prime numbers
# Arguments:
#   Input: n --- the number of primes to be generated
#   Output: p --- the array that contains the arrays
#
```

```
primearray:=proc(n,p)
   local i, k, count;

   p:=array(1..n);          # open the array to store output data
   count:=0;                # initialize the count
   k:=2;                    # the first integer to be examined

   while count < n do       # repeat until n primes are counted
      if isprime(k) then    # check if k is a prime
         count:=count+1;    # if so, count it
         p[count]:=k        # and store it in array p
      fi;
      k:=k+1;               # prepare the next integer
   od;

   print(`I got them`)      # print out a success message
end;
```

```
> read(`a:priarray.txt`);
```

primearray := **proc**(n, p)
local i, k, *count*;
p := array($1..n$) ;
count := 0 ;
k := 2 ;
while *count* $< n$ **do**
if isprime(k) **then** *count* := *count* + 1 ; $p_{count} := k$ **fi** ; $k := k + 1$
od;
print(`*I got them*`)
end

```
> primearray(50,p);
```

I got them

```
> seq(p[i],i=1..50);
```

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179,
181, 191, 193, 197, 199, 211, 223, 227, 229

```
> p[25]*p[12];
```

3589

Remark: In this example, we don't print out solutions inside the program. Instead, we use an argument p as output, which is an array of the data we want. After running the program, the data is on surface of Maple and can be referenced for other purpose.

3.1.6 Example: Identifying the maximum entry of an array

Find the maximum number in an number array: 2, -5, 9, π

```
#
# program to find the index of the maximum entry of a number array
# Input: m --- leading index of the array
#        n --- (>m) ending index of the array
#        s --- the array of numbers
# Output: index_max --- the index of the maximum entry
#         value_max --- the value of the maximum entry
#
find_max:=proc(m, n, s)
   local i, value_max, index_max;

   value_max:=evalf(s[m]);               # assume the first one is max
   index_max:=m;

   for i from m+1 to n do                # check the remaining entries 1 by 1
      if evalf(s[i]) > value_max then    # if a bigger value is found ...
         value_max:=evalf(s[i]);         #   update the maximum value
         index_max:=i;                   #   update the index of the max
      fi;
   od;

   print(`The maximum value:`);
   print( value_max );
   print(`The index of the maximum value`);
   print( index_max );

end;
```

```
> data:=array(1..7,
> [2, -5, 9, Pi, sqrt(5), -exp(1), sin(4*Pi/5)]);
```

$$data := \left[2,\ -5,\ 9,\ \pi,\ \sqrt{5},\ -e,\ \frac{1}{4}\sqrt{2}\sqrt{5-\sqrt{5}}\right]$$

```
> read(`a:findmax.txt`);
```

$find\_max :=$ **proc**(m, n, s)
local i, $value\_max$, $index\_max$;
$value\_max := \mathrm{evalf}(s_m)$;
$index\_max := m$;
for i **from** m **to** n **do**
if $value\_max < \mathrm{evalf}(s_i)$ **then** $value\_max := \mathrm{evalf}(s_i)$; $index\_max := i$ **fi**
od;
print(‘*The maximum value :* ‘);
print($value\_max$);
print(‘*The index of the maximum value*‘);
print($index\_max$)
end

```
> find_max(1,7,data);
```

The maximum value :

9.

The index of the maximum value

3

3.2 Some statistical measurements

3.2.1 Example 1. Range

Let $x_1, x_2, \cdots, x_n$ be a sequence of data. The range R is defined as $\max_i\{x_i\} - min_i\{x_i\}$. Write a program to calculate the range for input n and the data sequence.

```
#
# program to find the range of a number array, defined as
#      the maximum minus the minimum
#
# Input: m --- leading index of the array
#        n --- (>m) ending index of the array
#        x --- the array of numbers
# Output: The range
#
Range:=proc(m, n, x)
   local i, value_max, value_min;

   value_max:=evalf(x[m]);             # At beginning, assume the first
   value_min:=evalf(x[m]);             #   one is max and min

   # find the maximum

   for i from m to n do                # check the remaining entries
      if evalf(x[i]) > value_max then  # if a bigger value is found ...
         value_max:=evalf(x[i]);       #   update the maximum value
      fi;
   od;

   # find the minimum

   for i from m to n do                # check the remaining entries
      if evalf(x[i]) < value_min then  # if a smaller value is found ...
         value_min:=evalf(x[i]);       #   update the maximum value
      fi;
   od;

   # calculate and output the range

   value_max - value_min               # output of the range

end;
```

```
> read(`a:range.txt`);
```

```
Range := proc(m, n, x)
local i, value_max, value_min;
    value_max := evalf(x_m);
    value_min := evalf(x_m);
    for i from m to n do if value_max < evalf(x_i) then value_max := evalf(x_i) fi
    od;
    for i from m to n do if evalf(x_i) < value_min then value_min := evalf(x_i) fi od
    ;
    value_max − value_min
end
> a:=array(1..10,[2,45,-23,25,43,-26,89,-19,56,9]);
```

$$a := [2, 45, -23, 25, 43, -26, 89, -19, 56, 9]$$

```
> rang:=Range(1,10,a);
```

$$rang := 115.$$

3.2.2 Sorting

Suppose we have a sequence x_m, x_{m+1}, $\cdots$, x_n, and we want to rearrange the sequence to ascending order. We use m, instead of number 1, here as the leading index of the array for flexibility. In earlier examples we have seen that the array index may start from 0 rather than 1. The following process rearranges the array in ascending order..

step m: find the smallest entry among $x_m, x_{m+1}, \cdots, x_n$ and swap it to x_m.

step m+1: find the smallest entry among $x_{m+1}, x_{m+2}, \cdots, x_n$ and swap it to x_{m+1}.

step m+2: find the smallest entry among $x_{m+2}, x_{m+3}, \cdots, x_n$ and swap it to x_{m+2}.

$\cdots\cdots$

step n-1: find the smaller entry between x_{n-1} and x_n and swap it to x_{n-1}

(Question: Is there step n?)

Generally, we have a loop for $k = m, m+1, m+2, \cdots, n-1$,

step k: find the smallest entry among $x_k, x_{k+1}, \cdots, x_n$ and swap it to x_k.

Within step k, it is also necessary to have a loop to find the smallest entry in a subsequence. Therefore, we have a *nested loop*.

The program:

```
#
# Program that sort a sequence of data
#     x[m], x[m+1], ... , x[n]
# into ascending order
#
# input: m   ---  leading index of the sequence
#          n   ---  (>n) ending index of the sequence
#          x   ---  the data array with index range equal
#                   beyond [m,n]
#
sort_ascend:=proc(m,n,x)
   local k, j, tmp, value_min, index_min;

   if n < m then
      print(`starting index should not be larger than ending index`);
      RETURN()
   fi

   for k from m to n-1 do

      # find the value_min and index_min of   x[k],x[k+1],...,x[n]

      value_min:=evalf(x[k]);
      index_min:=k;

      for j from k+1 to n do
         if value_min > evalf(x[j]) then
            value_min:=evalf(x[j]);
            index_min:=j;
         fi;
      od;

      # swap to k-th entry if necessary

      if index_min > k then
         tmp:=x[index_min];
         x[index_min]:=x[k];
         x[k]:=tmp;
      fi;

   od;

   print(`ascending sorting finished`);

end;
```

```
> read(`a:sortascd.txt`):
> a:=array(2..10,[3,-4,9,8,-Pi,exp(1),sqrt(12),1/3,0]):
> sort_ascend(2,10,a);
```

ascending sorting finished

```
> seq(a[i],i=2..9);
```

$$-4,\ -\pi,\ 0,\ \frac{1}{3},\ e,\ 3,\ 2\sqrt{3},\ 8$$

3.3 Sieving

3.3.1 Sieving

Sieving is the process of striking out entries in a sequence

$$x_m, x_{m+1}, \cdots, x_n$$

that have (or do not have) a certain property.

One way to sieve is to have a shadow sequence, say

$$s_m, s_{m+1}, \cdots, s_n$$

all initialized with string "alive". If x_k is to be removed, assign $s_k :=$ "killed". At the end of the sieving, check those s_m, s_{m+1}, $\cdots$, s_n, if anyone, say $s_j =$ "alive", then the corresponding x_j survives the sieving.

3.3.2 Example: prime numbers by sieving

The problem is to find all prime numbers between 1 and n by striking out multiples of prime numbers.

The first prime number is 2, so we strike all multiples of 2. i.e. we mark numbers 2*3, 2*4, 2*5, $\cdots$ (up to n) as killed.

The next survived number is prime number 3, so we strike all multiples of 3. i.e. we mark 3*3, 3*4, 3*5, $\cdots$ (up to n) as killed. (Note we don't have to strike out 3*2, which is already killed)

The next survivor is 5 so we strike all multiples of 5, i.e., we mark 5*5, 5*6, 5*7, $\cdots$ (up to n) as killed. (Note, we don't have to kill 5*2, 5*3, 5*4, which were killed already).

Generally, the current prime number is k, we strike all multiple of $k$as killled. (Note, we don't have to kill k*2, k*3, ..., k*(k-1), which are killed already). Then we search the next survivor as next k, and strike out its multiples if $k\,k \le n$.

```
#
# Program that find all prime numbers within n by sieving
#
# input:  n --- the upper bound for searching
# output: p --- the array that contains all prime
#               numbers up to n
```

```
#
prime_sieve:=proc(n,p)
   local s, i, k, count, flag;

   # initialize the shadow array
   s:=array(1..n,[ seq("alive",i=1..n) ] );

   # 1 is not prime
   s[1]:="killed";

   # quick exit if n < 2
   if n < 2 then
      print(`There is no prime numbers in the given range`);
      RETURN()
   fi

   # sieve until k*k > n
   for k from 2 to n while k^2 <= n do

      # mark every i*k as killed if k is prime
      if s[k]="alive" then
         for i from k to n while i*k <= n do
            s[i*k]:="killed";
         od;
      fi;

   od;

   # count the number of primes found
   count:=0;
   for i from 1 to n do
      if s[i]="alive" then
         count:=count+1;
      fi;
   od;

   # open an array to store prime numbers
   p:=array(1..count);

   # initialize k as counter
   k:=0;
   for i from 1 to n do
      if s[i]="alive" then
         # when s[i] is `alive`, i is the k-th prime
         k:=k+1;
         p[k]:=i;
      fi;
   od;

   print(`The number of primes:`);
   count;

end;
```

Now let's find out all prime numbers from 1 to 1000.

```
> read(`a:prmsieve.txt`):
```

```
> prime_sieve(1000,p);
```

The number of primes :

168

```
> eval(p);
```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]

3.3.3 Example: Ulam's lucky numbers

From the list of positive integers 1, 2, 3, 4, $\cdots$, remove every second number, leaving 1, 3, 5, 7, 9, ... Since 3 is the first surviving number above 2 that has not been used as the "killer", we remove every 3rd number from the *remaining* numbers, yielding 1, 3, 7, 9, 13, 15, 19, 21, ... Now every 7th survivor is removed, leaving 1, 3, 7, 9, 13, 15, 21, ... Numbers that are never removed are considered to be "lucky". Write a program which prints the lucky numbers up to n.

```
#
#  Ulam's lucky numbers: For a sequence
#         1, 2, 3, ..., n
#  remove every 2nd number, it remains:
#         1, 3, 5, 7, 9, 11, 13, ...
#  then remove every 3rd number. Generally, after
#  removing every k-th number, let m be the first
#  surviving number that is larger than k. Update
#  k to be m and remove every k-th number in the
#  surviving sequence. This process is continued
#  until k is larger than the size of the surviving
#  sequence.
#
#  input:  n        --- the upper bound of searching
#  output: lucky    --- array that contains all lucky numbers up to n
#

ulumluck:=proc(n,lucky)

   local i,count,u,k,flag,survive;
   #
   #  initialize the array u.
   #     u[i]=`alive`  : i survives
```

```
   #     u[i]=`killed` : i removed.
   #
   u:=array(1..n);
   for i from 1 to n do
      u[i]:="alive"
   od;

   survive:=n;   # there are n survivors at beginning

   for k from 2 to n while survive > k  do

      #
      #  remove every k-th number if k is alive
      #
      if u[k]="alive" then
         count:=0;                   # initialize count
         for i from 1 to n do
            if u[i]="alive" then      # found a survivor
               count:=count+1;        # count the survivor
               if count=k then        # the k-th one?
                  u[i]:="killed";     # remove it
                  count:=0;           # reset count
                  survive:=survive-1; # one less survivor
               fi;
            fi;
         od;
      fi;

   od;

   #
   # record survivors in array lucky
   #
   lucky:=array(1..survive);   # open array lucky
   count:=0;                   # set count
   for i to n do
      if u[i]="alive" then     # found a survivor
         count:=count+1;       # count it
         lucky[count]:=i;      # record in lucky

      fi;
   od;

   print(`The Ulam's lucky numbers`);
   seq(lucky[i],i=1..count);   # print

end;
```

```
> read(`a:ulam.txt`):
```

```
> ulamluck(100,lucky);
```

The Ulam's lucky numbers

1, 3, 7, 9, 13, 15, 21, 25, 31, 33, 37, 43, 49, 51, 63, 67, 69, 73, 75, 79, 87, 93, 99

3.4 Projects with arrays

3.4.1 Pascal triangle

The coeficients of binomials $(a+b)^n$ can be calculated using the so called the Pascal triangle:

n	$(a+b)^n$	coeficients $c_0,\, c_1,\, \cdots\, c_n$
0	1	1
1	$a+b$	1 1
2	$a^2+2\,a\,b+b^2$	1 2 1
3	$a^3+3\,a^2\,b+3\,a\,b^2+b^3$	1 3 3 1
4	$a^4+4\,a^3\,b+6\,a^2\,b^2+4\,a\,b^3+b^4$	1 4 6 4 1
5	$a^5+5\,a^4\,b+10\,a^3\,b^2+10\,a^2\,b^3+5\,a\,b^4+b^5$	1 5 10 10 5 1
...	...	...

There are some obvious facts about the coeficients $c_0,\, c_1,\, c_2,\, \cdots,\, c_n$

- For each n, there are $(n+1)$ coeficients $c_0,\, c_1,\, c_2,\, \cdots,\, c_n$
- $c_0 = c_n = 1$.
- Let $d_0,\, d_1,\, d_2,\, \cdots,\, d_{k-1}$ be the coeficients of $(a+b)^{(k-1)}$, then

$$c_0 = 1,\;\; c_1 = d_0 + d_1,\;\; c_2 = d_1 + d_2,\;\; \cdots,\;\; c_{k-1} = d_{k-2} + d_{k-1},\;\; c_k = 1$$

Write a program, for input n output (1) the coeficient array c, **as an argument**, and (2) print out the Pascal triangle. You program should work like the following:

```
> read(‘a:pascal.txt‘):
> Pascal_triangle(10,c);
```

1
1, 1
1, 2, 1
1, 3, 3, 1
1, 4, 6, 4, 1
1, 5, 10, 10, 5, 1
1, 6, 15, 20, 15, 6, 1
1, 7, 21, 35, 35, 21, 7, 1
1, 8, 28, 56, 70, 56, 28, 8, 1

$$1,\ 9,\ 36,\ 84,\ 126,\ 126,\ 84,\ 36,\ 9,\ 1$$
$$1,\ 10,\ 45,\ 120,\ 210,\ 252,\ 210,\ 120,\ 45,\ 10,\ 1$$

```
> seq( c[j], j=0..10 );
```

$$1,\ 10,\ 45,\ 120,\ 210,\ 252,\ 210,\ 120,\ 45,\ 10,\ 1$$

3.4.2 Means

For a sequence of real numbers

$$x_m,\ x_{m+1},\ x_{m+2},\ \cdots,x_n$$

various means are defined as follows:

(i) Arithmetic mean A:

$$A = \sum_{i=m}^{n} \frac{x_i}{n-m+1}$$

(ii) Geometric mean G:

$$G = (\prod_{i=m}^{n} x_i)^{(\frac{1}{n-m+1})}$$

(iii) Harmonic mean H:

$$\frac{1}{H} = \frac{\sum_{i=m}^{n} \frac{1}{x_i}}{n-m+1}$$

Write a program that, for input m, n, x; print out all three means. Your program should work like

```
> x:=array(0..10,[2,5,9,12,21,33,18,8,11,15,17]):
> read(`a:means.txt`):
> Mean(0,10,x);
```

The arithmetic mean :
13.72727273
The geometric mean :
11.05802293
The harmonic mean :
8.033176344

```
> y:=array(1..9,[3,9,12,43,31,18,24,38,5]):
> Mean(1,9,y);
```

The arithmetic mean :
20.33333333

$$\textit{The geometric mean}:$$
$$14.86548789$$
$$\textit{The harmonic mean}:$$
$$9.924686319$$

We can also calculate the means of the binomial coeficients:

```
> Pascal_triangle(12,c):
```

$$1$$
$$1, 1$$
$$1, 2, 1$$
$$1, 3, 3, 1$$
$$1, 4, 6, 4, 1$$
$$1, 5, 10, 10, 5, 1$$
$$1, 6, 15, 20, 15, 6, 1$$
$$1, 7, 21, 35, 35, 21, 7, 1$$
$$1, 8, 28, 56, 70, 56, 28, 8, 1$$
$$1, 9, 36, 84, 126, 126, 84, 36, 9, 1$$
$$1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1$$
$$1, 11, 55, 165, 330, 462, 462, 330, 165, 55, 11, 1$$
$$1, 12, 66, 220, 495, 792, 924, 792, 495, 220, 66, 12, 1$$

```
> Mean(0,12,c);
```

$$\textit{The arithmetic mean}:$$
$$315.0769231$$
$$\textit{The geometric mean}:$$
$$78.51839612$$
$$\textit{The harmonic mean}:$$
$$5.872498536$$

3.5 Exercises

1. **Fibonacci array**

   Write a program that, for input n, output an array of Fibonacci sequence

   $$F_0, F_1 \cdots F_n$$

   Do not print out the solution inside the program. Instead, use an argument as output and use seq to show results after running the program.

   Sample results

   ```
   > read(`a:fibarray.txt`):
   > n:=40:
   ```

```
> fibarray(40,f);
```

Fibonacci sequence is generated

```
> seq(f[k],k=0..n);
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986, 102334155

2. **Inner product (or dot product)**

   A n-vector is defined as an array of n numbers

$$(v_1, v_2, \cdots, v_n)$$

   such as

$$a = (1, 3, 5, -3, 2), \quad b = (9, -5, 2, \sqrt{7}, \pi)$$

   The inner product (or dot product) of two n-vectors

$$a = (a_1, a_2, \cdots, a_n), \quad \text{and} \quad b = (b_1, b_2, \cdots, b_n)$$

$$a \cdot b = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

   Write a program, for input n, a, and b, print out the value of the their inner product.

   Sample results

```
> read(‘a:dotprod.txt‘):
> n:=5:
> a:=array(1..n, [1,3,5,-3,2]);
```

$$a := [1, 3, 5, -3, 2]$$

```
> b:=array(1..n, [9, -5, 2, sqrt(7), Pi]);
```

$$b := \left[9, -5, 2, \sqrt{7}, \pi\right]$$

```
> dotprod(a,b,n);
```

The dot product is

2.345931375

3. **Deck list**

   Write a program that defines an array of 52 entries, each entry contains the the rank and suit of a card in a standard deck. That is, each entry is an array of two entries, the first entry is the rank, and the second entry the suit. The deck must be in the order of Example 3, Lecture Note 02. Write a program using do loops instead of 52 line program.

   (Hint: Define *deck* as a 52-entry array. deck[1] is defined as a two entry array, to be referenced as deck[1][1] and deck[1][2], such that deck[1][1]=ace, deck[1][2]=heart.)

   Sample results

```
> read(‘a:decklist.txt‘):
```

```
> decklist(deck):
> eval(deck);
```

[[*ace*, *heart*], [2, *heart*], [3, *heart*], [4, *heart*], [5, *heart*], [6, *heart*], [7, *heart*], [8, *heart*], [9, *heart*], [10, *heart*], [*jack*, *heart*], [*queen*, *heart*], [0, *heart*], [*ace*, *spade*], [2, *spade*], [3, *spade*], [4, *spade*], [5, *spade*], [6, *spade*], [7, *spade*], [8, *spade*], [9, *spade*], [10, *spade*], [*jack*, *spade*], [*queen*, *spade*], [0, *spade*], [*ace*, *diamond*], [2, *diamond*], [3, *diamond*], [4, *diamond*], [5, *diamond*], [6, *diamond*], [7, *diamond*], [8, *diamond*], [9, *diamond*], [10, *diamond*], [*jack*, *diamond*], [*queen*, *diamond*], [0, *diamond*], [*ace*, *club*], [2, *club*], [3, *club*], [4, *club*], [5, *club*], [6, *club*], [7, *club*], [8, *club*], [9, *club*], [10, *club*], [*jack*, *club*], [*queen*, *club*], [0, *club*]]

```
> eval(deck[18]);
```

[5, *spade*]

```
> eval(deck[37]);
```

[*jack*, *diamond*]

4. **Minimum**

   Write a program that identifies the mininum entry of an array by printing out the value of the maximum entry and its index. Use the data in Example 2 to test your program.

5. **Standard deviation**

   Write a program to calcaulate the standard deviation

$$s = \sqrt{\sum_{i=1}^{n} \frac{(x_i - \sigma)^2}{n}}$$

   where σ is the average of x_1, x_2, $\cdots$, x_n, i.e.:

$$\sigma = \sum_{i=1}^{n} \frac{x_i}{n}$$

6. **Median and quartiles**

   Let

$$x_m, x_{m+1}, \cdots, x_n$$

   be a sequence of $n - m + 1$ real numbers in *ascending* order. The leading index m can be 0, 1 or any other integer. The **median** this sequence is defined as follows:

   case 1. if $n - m + 1$ is odd, then the median is the middle term at index $\frac{n+m}{2}$.

   case 2. if $n - m + 1$middle-left term and the middle-right term with indices $\frac{n+m-1}{2}$ and $\frac{n+m+1}{2}$ respectively.

The median is also called the **second quartile**.

The **first quartile** and the **third quartile** are defined as follows:

case 1: *if* $n+m-1$ *is even*, then the sequence can be devided into the first half sub-sequence

$$x_m, \cdots, x_{\frac{n+m-1}{2}}$$

and the second half sub-sequence

$$x_{\frac{n+m+1}{2}}, \cdots, x_n.$$

The **first quartile** is the median of the first half sub-sequence and the **third quartile** is the median of the second half sub-sequence

case 2: *if n-m+1 is odd*, then there is a middle term $x_{\frac{n+m}{2}}$ The **first quartile** is defined as the median of the sub-sequence

$$x_m, \cdots, x_{\frac{n+m}{2}-1}$$

and the **third quartile** is defined as the median of the sub-sequence

$$x_{\frac{n+m}{2}+1}, \cdots, x_n.$$

Write a program, for any given sequence

$$x_m, x_{m+1}, \cdots, x_n$$

print out the three quartiles. Your program should accept input m, n, and array x, rearrange it into ascending order, and find quartiles.

Hint: You may consider the following steps:

Input: m, n, x

(1) re-arrange the array x in ascending order

(2) according to the odd/even number $n-m+1$

- find the median (2nd quartile)
- find the index range $m1$ and $n1$ of the first sub-sequence
- find the index range $m2$ and $n2$ of the second sub-sequence

(3) according to odd/even type of $n1-m1+1$ and $n2-m2+1$, find the first quartile and the third quartile respectively.

Sample results

```
> read(`c:/zeng/teach/340/quartile.txt`):
> x:=array(0..10,[23,12,34,87,25,10,5,19,65,29,71]):
> quartile(x,0,10);
```

The sequence in ascending order

$$5, 10, 12, 19, 23, 25, 29, 34, 65, 71, 87$$

The 1st subsequence

5, 10, 12, 19, 23

The 2nd subsequence

29, 34, 65, 71, 87

The 1st, 2nd, and 3rd quartiles :

12
25
65

```
> y:=array(1..14,[22,11,33,44,51,62,12,81,37,19,9,20,18,5]);
```

$$y := [22, 11, 33, 44, 51, 62, 12, 81, 37, 19, 9, 20, 18, 5]$$

```
> quartile(y,1,14);
```

The sequence in ascending order

5, 9, 11, 12, 18, 19, 20, 22, 33, 37, 44, 51, 62, 81

The 1st subsequence

5, 9, 11, 12, 18, 19, 20

The 2nd subsequence

22, 33, 37, 44, 51, 62, 81

The 1st, 2nd, and 3rd quartiles :

12
21.0
44

7. **Partial amnesty**

In the central prison of Sikinia there are n cells numbered from 1 to n, each occupied by a single prisoner. The state of each cell can be changed from closed to open and vice versa by a half-turn of the key. To celebrate the Centennial Anniversary of the Republic it was decided to grant a partial amnesty. The president sent an officer to the prison with the instruction:

```
for i from 1 to n do
    turn the keys of cells i, 2*i, 3*i, ... (up to n)
od;
```

A prisoner was freed if *at the end* his/her door was open. Which prisoners are were set free? Write a program that carry out the amnesty for n=100, 200, 500.

(Hint: The door can be either 'open' or 'closed'. Each time the key is turned, the door becomes 'closed' if it is open, or becomes 'open' if it is 'closed'. A door may be set open or closed many times)

Sample results

```
> read('a:amnesty.txt'):
> amnesty(100,free);
```

The number of prisoners freed

10

The following prisoners were set free:

```
> eval(free);
```

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

```
> amnesty(200,free);
```

The number of prisoners freed

14

```
> eval(free);
```

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]

```
> amnesty(500,free);
```

The number of prisoners freed

22

```
> eval(free);
```

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169,
196, 225, 256, 289, 324, 361, 400, 441, 484]

8. **The Josephus problem**

During the Jewish rebellion against Rome (A.D. 70) 40 Jews were caught in a cave. In order to avoid slavery they agreed upon a program of mutual destruction. They would stand in a circle and number themselves from 1 to 40. Then every seventh person was to be killed until only one was left who would commit suicide. The only survivor, Flavius Josephus, who did not carry out the last step, became a historian.

Josephus problem: n persons are arranged in a circle and numbered from 1 to n, Then every k-th person is removed, with the circle closing up after every removal. Print the list of persons in the order of being removed. What is the number f(n) of the last survivor?

Write a program to

(1) Print out the sequence of persons in the order of being killed.

(2) Find out where Josephus is.

Sample results

```
> read('a:josephus.txt'):
> Josephus(40,7);
```

People were killed in the following order

7, 14, 21, 28, 35, 2, 10, 18, 26, 34, 3, 12, 22, 31, 40, 11, 23, 33, 5, 17, 30,
4, 19, 36, 9, 27, 6, 25, 8, 32, 16, 1, 38, 37, 39, 15, 29, 13, 20

Josephus is at

24

Suppose we have 14 students registered for this class, so we have 15 people (including the instructor) in the classroom, if we play this game:

```
> Josephus(15,7);
```

People were killed in the following order

7, 14, 6, 15, 9, 3, 13, 11, 10, 12, 2, 8, 1, 4

Josephus is at

5

The fifth person is the final survivor. Or, if every 5th person is removed,

```
> Josephus(15,5);
```

People were killed in the following order

5, 10, 15, 6, 12, 3, 11, 4, 14, 9, 8, 13, 2, 7

Josephus is at

1

The first person will survive.

9. **Weighted means:**

Let

$$\lambda = (\lambda_m, \lambda_{m+1}, \cdots, \lambda_n)$$

be a vector of positive numbers such that

$$\sum_{i=m}^{n} \lambda_i = 1$$

then the weighted arithmetic and geometric means associated with λ are defined as

(i) weighted arithmetic mean:

$$A_\lambda = \sum_{i=m}^{n} \lambda_i \, x_i = \lambda_m \, x_m + \lambda_{m+1} \, x_{m+1} + \cdots + \lambda_n \, x_n$$

(ii) weighted geometric mean:

$$G_\lambda = \prod_{i=m}^{n} {x_i}^{\lambda_i} = \left(({x_m}^{\lambda_m}) \left({x_{m+1}}^{\lambda_{m+1}}\right) \cdots \left({x_n}^{\lambda_n}\right) \right)$$

Let the weights

$$\lambda_m, \lambda_{m+1}, \cdots, \lambda_n$$

be defined as follows:

$$\lambda_m = \frac{1}{2}, \lambda_{m+1} = \frac{1}{4}, \cdots, \lambda_i = \frac{\lambda_{i-1}}{2}, \cdots, \lambda_{n-1} = \frac{\lambda_{n-2}}{2}, \lambda_n = \lambda_{n-1}$$

Write a program that, for input m, n, x, calculate both weighted means. Use your program to calculate the weighted means of the two sequences in Project 2 above.

Sample results

```
> read(`a:wmeans.txt`):
> Weighted_means(0,10,x);
```

The weighted arithmetic mean

5.521484376

The weighted geometric mean

3.933248559

```
> Weighted_means(1,9,y);
```

The weighted arithmetic mean

9.54296875

The weighted geometric mean

6.309771865

```
> Weighted_means(0,12,c);
```

The weighted arithmetic mean

64.87329102

The weighted geometric mean

6.551390952

10. **The power of binomials**

Write a program that, for input a, b, n, calculate $(a+b)^n$ using the Pascal triangle and the resulting coeficients of binomials. Your program should have the structure like:

step 1, calculate

$$c_0,\, c_1,\, \cdots,\, c_n$$

using Pascal triangle

step 2, calculate and print out

$$c_0\, a^n + c_1\, a^{(n-1)}\, b + c_2\, a^{(n-2)}\, b^2 + c_3\, a^{(n-3)}\, b^3 + \cdots + c_{n-1}\, a\, b^{(n-1)} + c_n\, b^n$$